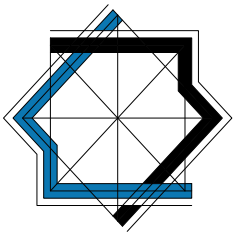
UNIVERSIDAD DE GRANADA



TRABAJO FIN DE MÁSTER

MÁSTER EN MATEMÁTICAS

Algoritmos en grafos paso a paso

Autor
José González Ibáñez

Tutor: Pedro A. García Sánchez
Departamento de Álgebra

Febrero 2019

Índice general

Capítulo 1

Motivación

Imaginemos el caso de un repartidor de pan, el cuál tiene una clientela fija a la que debe llevar su pedido cada mañana. Este repartidor posiblemente sea un autónomo que se deba costear el combustible de su propio vehículo y a quien el reparto le priva de un tiempo vital que retrasa el resto de sus tareas. Pues bien, este trabajador debe optimizar al máximo sus viajes para que el coste económico y temporal de su reparto sea mínimo, es decir, buscar la ruta más eficiente de forma que abastezca a todos sus clientes.

Imaginemos ahora que, por las condiciones de la carretera, el tiempo que tardaría el repartidor en bicicleta es el mismo que tardaría con un vehículo motorizado. Imaginemos también, que las carreteras que debe recorrer están dotadas de un carril bici. Al margen de la reducción del gasto por no usar combustible en esta segunda situación, hay una pequeña diferencia entre estos dos problemas que cambia considerablemente la obtención de la ruta más eficiente: en el segundo caso, la carretera puede ser recorrida en ambos sentidos.

A pesar de que las situaciones anteriores pueden parecer simples o de escasa repercusión, la Teoría de Grafos es utilizada para el estudio de problemas de ese tipo, similares en esencia, no en contexto.

La Teoría de Grafos se ha desarrollado profundamente a lo largo de la historia [2], hasta convertirse en una herramienta fundamental en diversas áreas de conocimiento que van desde la matemática, la química o la informática, hasta la educación o la traducción de idiomas y, en general, aparece en cualquier situación en la que se pretenda optimizar el reparto de un recurso entre diversos puntos o estudiar la conexión entre los elementos de un conjunto.

- Orientación de las carreteras de zonas urbanas.
- Diseño de redes eléctricas, de computación, de comunicación, etc.
- Distribución de mercancías.
- Construcción de edificios para abastecer a varios núcleos poblacionales.
- Análisis de redes sociales.

- Big Data.
- Búsqueda de los isómeros de diferentes compuestos químicos.
- Establecer las secuencias de las moléculas de ADN.

1.1. Motivación personal

La historia [2] y gran aplicabilidad de la Teoría de Grafos en la vida real es una de las razones que me ha llevado a la profundización en su estudio, pues al margen de lo mencionado anteriormente, los grafos también son una gran herramienta didáctica, algo que me ayudará en un futuro, puesto que mis intenciones son convertirme en docente en matemáticas. Los grafos nos permiten visualizar, reescribir, modelizar situaciones reales para simplificar su estudio, promoviendo en los estudiantes el razonamiento lógico-matemático y los pensamientos inductivo y combinatorio. Además, también se utilizan para obtener estrategias vencedoras en diversos juegos matemáticos, algunos tan famosos como el Laberinto de Rouse Ball o las Torres de Hanoi. Finalmente, otra de las razones por las que me embarqué en este tema, es por su parte práctica. Durante la carrera de matemáticas descubrí un gran interés en la programación informática, el cuál no se vio satisfecho durante el grado. Por ello, buscaba realizar un trabajo de fin de Máster que me permitiera ampliar mis conocimientos en este campo, haciéndome crecer así como programador.

Capítulo 2

Introducción

En una primera aproximación, y sin conocer aún terminología precisa, podríamos definir la Teoría de Grafos como la rama de la Matemática Discreta encargada de estudiar la conectividad entre los elementos de un conjunto.

La Teoría de Grafos surgió a mediados del siglo XVIII, cuando Leonard Euler publicó su artículo *'Solutio problematis ad geometrian situs pertinentis'* con la solución al problema de los puentes de Königsberg, el cual procedemos a enunciar. Königsberg era una ciudad de la antigua Prusia Oriental que, atravesada por el río Pregel, quedaba dividida en cuatro porciones de tierra, todas ellas unidas por siete puentes, como se muestra en la Figura 2.1.

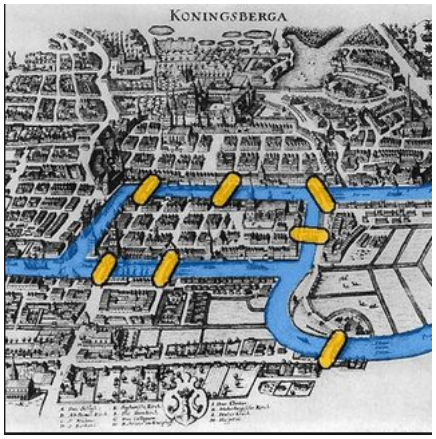


Figura 2.1. Mapa de la ciudad de Königsberg.

"¿Es posible recorrer todos los puentes una sola vez y volver al punto de partida?" Esta es la pregunta que tanto habitantes como turistas se realizaban. Euler fue la primera persona que demostró formalmente que no se podía realizar tal recorrido. Para ello, esquematizó el problema matemáticamente quedándose sólo con la información relevante y obviando características como la longitud de los puentes o el área de las porciones terrestres, entre otras.

Así, identificó cada pedazo de tierra como un punto y cada puente como líneas que unían tales puntos, una idea sencilla pero revolucionaria para su época,

reduciendo el problema al esquema que observamos en la Figura 2.2, que se conoce como *grafo*.

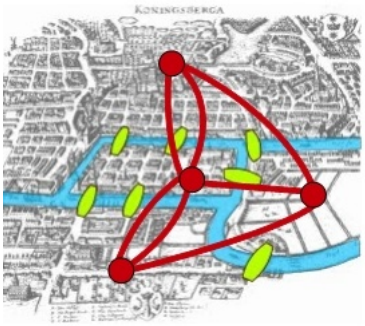


Figura 2.2. Grafo sobre la ciudad de Königsberg.

Así pues, a raíz de este inesperado acertijo, surgió la Teoría de Grafos, a la cuál realizaron importantes aportaciones grandes matemáticos como Vandermonde, Hamilton, Cauchy, Cayley, etc.

Uno de los objetivos de esta memoria ha sido elaborar unos apuntes que permitan al alumnado introducirse en la Teoría de Grafos, explicando los tipos de grafos, los elementos que los componen y algunos de los resultados más importantes sobre ellos. Para ello, durante todo este proyecto nos hemos apoyado en [3], [5], [10] y [11]. El otro objetivo principal es la creación de un repositorio [7] con la implementaciones de estos algoritmos. Estas implementaciones tienen la posibilidad de mostrar cada uno de los pasos de la ejecución de los algoritmos estudiados. Para la representación de los elementos de la clase *grafo* han sido necesarias las librerías *graphviz* e *ipywidgets*, esta última para poder mostrar el funcionamiento de las funciones implementadas paso a paso con botones.

El enlace para el repositorio mencionado, donde podemos encontrar un tutorial de uso de la clase *grafo* (inspirada en [6]) y de los algoritmos implementados sobre ella es:

`https://github.com/lmd-ugr/Algoritmos-sobre-grafos`

Comenzaremos este trabajo con un primer capítulo donde describiremos los elementos que componen los grafos, los distintos tipos en que se clasifican según sus características y propiedades o resultados que se deducen sobre ellos.

En el segundo capítulo de esta memoria, explicaremos dos tipos de grafos muy importantes: los grafos de Euler y de Hamilton. Incluiremos los algoritmos que nos permiten identificarlos y extraer sus elementos más importantes y añadiremos ejecuciones paso a paso para facilitar su comprensión.

Continuaremos con un tercer capítulo que abarcará las sucesiones gráficas y la coloración de grafos. El teorema de los cuatro colores y el teorema de Hawel-Hakimi, junto con la implementación del algoritmo que aporta su demostración, serán las piezas clave de este apartado.

Finalmente, cerraremos el trabajo abordando algunas de las estructuras más importantes dentro de la Teoría de Grafos, los árboles. Estudiaremos algunos de

los diferentes tipos en que se pueden clasificar, cómo obtenerlos y cómo conseguir algunos de sus elementos más útiles.

Capítulo 3

Introducción a la Teoría de Grafos

3.1. Nociones básicas

Definición 1. Un *grafo* G es un par (V,E), donde V es el conjunto de vértices (también llamados nodos) y E, el conjunto de aristas (también llamadas lados), junto con una aplicación Φ_G denominada aplicación de incidencia, la cual lleva cada arista de E al conjunto de vértices de V en los que incide:

$$\Phi_G : E \to \{\{u,v\} : u,v \in V\}.$$

Definición 2. Un grafo $G = (V,E)$ se denomina *vacío* si $E = \emptyset$, es decir, si G no contiene ninguna arista, solamente vértices.

Definición 3. Se define el *grafo trivial* como el grafo vacío que contiene un sólo vértice.

Mostremos ahora un ejemplo de grafo. En la Figura 3.1 observamos que $V = \{1,2,3,4\}$ y $E = \{a,b,c,d\}$.

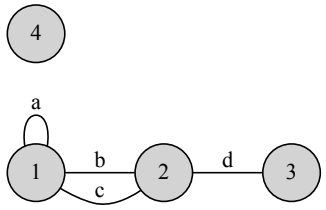


Figura 3.1

La aplicación de incidencia quedaría como sigue:

$$\Phi_G(a) = \{1\},\ \ \Phi_G(b) = \Phi_G(c) = \{1,2\},\ \ \Phi_G(d) = \{2,3\}.$$

Podemos resaltar algunas características en este grafo. En primer lugar, tenemos una arista, a, que incide solamente en un vértice, a este tipo de aristas se les llama

lazos. Por otro lado, observamos que dos de las aristas del grafo, b y c, inciden sobre los mismos vértices, en este caso decimos que tales aristas son *paralelas*.

Es importante añadir que algunos autores llaman a este tipo de grafos, *multigrafos*, mientras que los *grafos* son aquellos que no poseen ni lazos ni aristas paralelas. En esta memoria, utilizaremos los términos *grafo* y *grafo simple* en lugar de *multigrafo* y *grafo*, respectivamente.

Podemos hacer una última observación y es que, al hablar de las aristas, decimos que inciden sobre los vértices pero no mencionamos cuál es el nodo origen o el nodo final. Esto es así porque, salvo que se indique, se considera que las aristas se pueden recorrer en ambos sentidos. Así pues, esta observación nos lleva a introducir un nuevo tipo de grafo, el *grafo dirigido* o *digrafo*, donde las aristas parten del vértice origen al vértice final siendo, por tanto, recorridas en un solo sentido.

Definición 4. Un *grafo dirigido* o *digrafo* D es un par (V,E), con V el conjunto de vértices y E el conjunto de aristas, junto con dos aplicaciones:

$$O_D : E \to \{u : u \in V\},$$
$$F_D : E \to \{v : v \in V\}.$$

Es intuitivo pensar que la aplicación O_D, también llamada *aplicación dominio*, devuelve el nodo origen o de partida de la arista en cuestión, mientras que la aplicación F_D, denominada *aplicación codominio*, devuelve el nodo final o de llegada de la arista referida.

De esta forma, volviendo a la Figura 3.1, podemos transformarla en un digrafo, por ejemplo, como indicamos en la Figura 3.2.

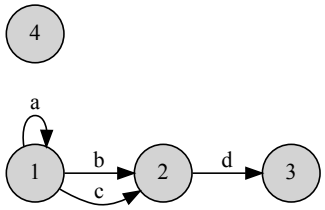


Figura 3.2. Digrafo.

Las aplicaciones dominio y codominio de algunas de sus aristas quedarían entonces como sigue:

$$O_D(a) = F_D(a) = \{1\},\ \ O_D(d) = F_D(b) = \{2\},\ \ O_D(b) = O_D(c) = \{1\}.$$

Definición 5. Sea $G = (V,E)$ un grafo simple. Definimos el *complementario* del grafo G como el grafo simple $\overline{G} = (V,\overline{E})$ que tiene el mismo conjunto de vértices y, como conjunto de aristas, aquellas que no aparecen en G.

Definición 6. Dado un grafo con conjunto de vértices V, se dice *ponderado*, si se especifica un peso para cada una de sus aristas, esto es, una función $w : V \to \mathbb{R}$.

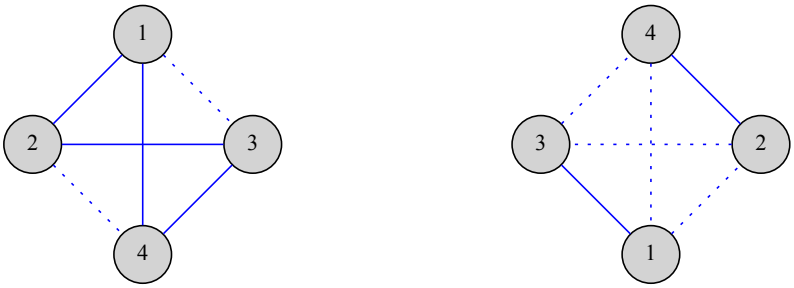


Figura 3.3. Grafos complementarios.

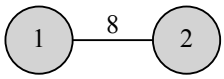


Figura 3.4. Grafo ponderado.

En el ejemplo de la Figura 3.4, encontramos un grafo ponderado con una única arista, cuyo peso es 8.

Definición 7. Sea G un grafo. Decimos que G es un *grafo plano* si admite una representación plana, es decir, una representación donde vértices y aristas quedan todos en un plano cumpliendo que dos aristas distintas no se cortan.

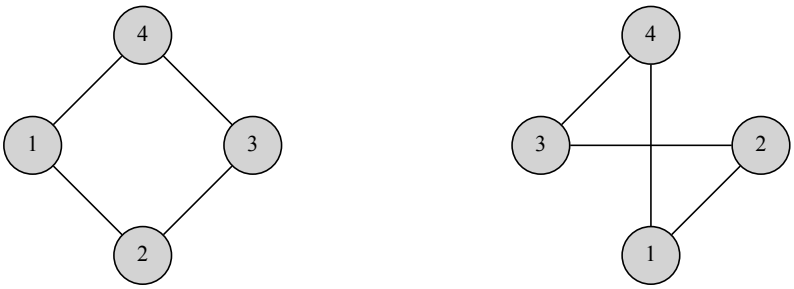


Figura 3.5. Dos representaciones distintas de un mismo grafo.

Podemos observar en la Figura 3.5 un mismo grafo representado de dos formas distintas, al ser la representación de la izquierda, una representación plana, entonces podemos decir que dicho grafo es plano.

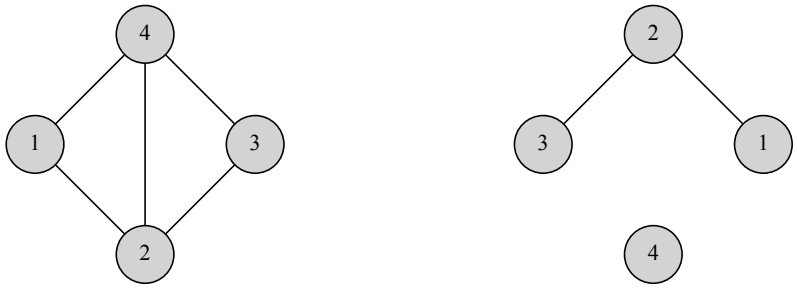


Figura 3.6. A la izquierda un grafo G y a la derecha un subgrafo suyo.

Definición 8. Sea H un grafo con conjunto de vértices V' y conjunto de aristas E'. Decimos que H es un *subgrafo* de $G = (V, E)$ si $V' \subset V$, $E' \subset E$ y se cumple que $\Phi_G(e) = \Phi_H(e)$ para todo $e \in E'$, siendo Φ_G y Φ_H las aplicaciones de incidencia de los grafos G y H respectivamente.

- Se dice que el grafo G contiene al grafo H si H es un subgrafo de G o es isomorfo (véase Sección 3.2) a algún subgrafo de G.
- Sea $H = (V', E')$ un grafo, H se denomina *subgrafo inducido o completo* del grafo $G = (V, E)$ si posee todos los lados adyacentes a los nodos de V', es decir, dada una arista $e \in E$ tal que $\Phi_G(e) \subset V'$, se verifica que $e \in E'$.

Para continuar, vamos a definir algunos conceptos y elementos que encontramos en los grafos. Acompañaremos estas definiciones con ejemplos sobre el grafo de la Figura 3.1.

- Dos nodos se dicen *adyacentes* si están conectados por una arista. Cuando sobre un vértice no incide ninguna arista, a éste se le denomina vértice *aislado*. En la Figura 3.1 observamos que, por ejemplo, los nodos 1 y 2 son adyacentes mientras que los nodos 1 y 3 no lo son. Además, hay un nodo aislado, el número 4.
- Se denomina *grado de un vértice* y lo denotaremos como $\mathrm{gr}(v)$ al número de aristas que inciden sobre él. Así, si uno de los vértices posee un lazo, este incrementará en dos el grado del vértice donde incide. En la Figura 3.1 $\mathrm{gr}(1) = 4$, $\mathrm{gr}(2) = 3$, $\mathrm{gr}(3) = 1$ y $\mathrm{gr}(4) = 0$. Es importante mencionar que, cuando se trata de un grafo dirigido, diferenciamos entre *grado de entrada* de un vértice para referirnos al número aristas que llegan a ese vértice y *grado de salida* de un vértice para referirnos al número de aristas que parten de dicho vértice.

- Se define la *sucesión de grados* de un grafo como $D(G) = \{D_k\}_{k\in\mathbb{N}}$ donde D_k es el número de vértices de G que tienen grado k. Así pues, el grafo de la Figura 3.1 tiene la siguiente sucesión de grados:

$$1\ 1\ 0\ 1\ 1.$$

- Denominamos *tamaño de un grafo* al número de aristas que lo componen, mientras que llamamos *orden de un grafo* al número de nodos que posee. De esta forma, el *grafo trivial* tiene tamaño 0 y orden 1, mientras que el grafo de la Figura 3.1 tiene tamaño y orden 4.

- Sea G un grafo, llamamos *camino de longitud n* entre los vértices v_0 y v_n a una secuencia de n aristas $e_1e_2\dots e_n$ y una secuencia de $n+1$ vértices $v_0v_1\dots v_n$ que cumplen que $\Phi_G(e_i) = \{v_{i-1}, v_i\}$ para todo $i \in \{1,\dots,n\}$. En tal caso, se dice que los vértices v_0 y v_n están conectados. En la Figura 3.1, la sucesión de aristas bd y la sucesión de vértices $1-2-3$ forman un camino de longitud 2 desde el vértice 1 hasta el vértice 3.

- Llamamos *recorrido* a un camino donde no se encuentran lados repetidos. El ejemplo anterior también es un recorrido.

- Se denomina *camino simple* a un recorrido que no contiene vértices repetidos salvo, eventualmente, los vértices inicial y final. Es decir, un camino en el que no se repiten ni lados ni nodos. De nuevo, el ejemplo anterior nos vale, es un camino simple.

- Definimos *camino cerrado* como un camino cuyo vértice inicial y vértice final son el mismo. En la Figura 3.1, un ejemplo de camino cerrado sería la secuencia de aristas bc junto con la secuencia de vértices $1-2-1$.

- Se llama *circuito* a un recorrido que empieza y acaba en el mismo nodo. Es decir, un camino cerrado sin aristas repetidas. El caso anterior también nos vale como ejemplo de un circuito.

- Denominamos *ciclo* a un camino cerrado sin vértices ni aristas repetidas. De nuevo, el ejemplo anterior es también un ciclo.

- Definimos la *distancia* entre dos vértices como la longitud del recorrido más corto entre ellos. La distancia entre dos vértices no conectados se toma como infinita. En la Figura 3.1, la distancia entre los vértices 1 y 3 es dos y la distancia entre el vértice 4 y cualquiera de los otros, es infinito.

- Se define el *diámetro* de un grafo como el máximo de todas las distancias de los vértices del grafo. En la Figura 3.1, el diámetro del grafo es infinito.

Una vez definidos estos elementos, podemos ahora enunciar algunas proposiciones acerca de ellos o sobre su existencia en cierto tipo de grafos, así como definir algún tipo más de grafo, como vemos a continuación.

Proposición 1. *Sea el grafo $G = (V, E)$ y sea $H = (V', E')$ un subgrafo suyo, se verifica entonces que*

$$\mathrm{gr}_H(v) \leq \mathrm{gr}_G(v) \quad \forall v \in V'.$$

Proposición 2. *Sea $G = (V, E)$ un grafo de n vértices. Se verifica que $\sum_{i=1}^{n} \mathrm{gr}(v_i) = 2l$ siendo l el número de aristas totales del grafo. Es decir, la suma de los grados de todos los vértices de un grafo es el doble del número total de aristas que contiene el grafo.*

Demostración. La demostración es muy simple. Dado que cada arista incide en dos vértices, al sumar los grados de cada vértice se ha contado cada arista por duplicado, por tanto, al final obtenemos el doble del número total de aristas. □

Proposición 3. *Sean $G = (V, E)$ un grafo y sean $u, v \in V$. Si existe un camino que conecte u y v, entonces existe un camino simple que conecte u con v.*

Proposición 4. *Dado un grafo $G = (V, E)$ y dados $u, v \in V$ dos vértices distintos, supongamos que existen dos caminos simples distintos conectando u y v, entonces existe un ciclo conectando u y v.*

Lema 1. *Sea $G = (V, E)$ un grafo finito verificando que $\mathrm{gr}(v) > 1$ para todo $v \in V$. Entonces existe un circuito en G y, por tanto, un ciclo.*

Demostración. Tomamos como vértice inicial un vértice cualquiera de G y lo llamamos v_0. Tenemos que $\mathrm{gr}(v_0) > 1$, entonces tomamos una arista incidente en v_0, que la denotamos como e_1. Cojamos ahora el otro extremo de e_1, vértice que denotamos por v_1. Si $v_0 = v_1$ se tiene que e_1 es un lazo y, por tanto, un circuito. En caso contrario, de nuevo, tenemos que $\mathrm{gr}(v_1) > 1$, por tanto, escojo una arista incidente en v_1 diferente de e_1, sea ésta e_2 y llamemos v_2 al otro vértice donde incide e_2 distinto de v_1. Entonces ya tenemos el camino simple determinado por la sucesión de vértices $v_0 v_1 v_2$ y la sucesión de aristas $e_1 e_2$. Siguiendo el proceso de esta forma, como el grafo es finito, en algún momento acabaremos visitando un vértice por el que ya hemos pasado, obteniendo así el circuito buscado. □

Definición 9. Sea $G = (V, E)$ un grafo. Se dice que G es *conexo* si para todo $u, v \in V$, existe al menos un camino que conecte u y v.

En el ejemplo de la Figura 3.1, no tenemos un grafo conexo, dado que no existe ningún camino que conecte el vértice 4 con cualquiera de los otros. Podemos entonces definir la relación Ω para los vértices del grafo como: $u \in V$ está relacionado con $v \in V$, $u \Omega v$, si están conectados, es decir, si existe un camino entre ambos vértices. Veamos que esta relación es de equivalencia.

- Reflexividad: $u \Omega u$ ya que todo vértice está conectado consigo mismo por un camino de longitud 0.
- Simetría: Si $u \Omega v$, entonces $v \Omega u$ ya que, en un grafo, si existe un camino entre u y v, ese mismo camino, recorrido en sentido inverso, nos lleva de v a u.

- Transitividad: Si $u\Omega v$ y $v\Omega w$, entonces $u\Omega w$ ya que si hay un camino que conecta u con v y otro camino conectando v con w, basta componer ambos caminos para tener conectados u y w.

De esta forma, utilizando la relación Ω, podemos caracterizar de una forma distinta la conexión de un grafo.

Definición 10. Sea G un grafo, se dice que G es *conexo* si el conjunto cociente por la relación Ω está compuesto de un solo elemento. En caso contrario, el grafo se dice *disconexo*. Además, definimos cada *componente conexa* de G como el subgrafo completo generado por los vértices de cada clase de equivalencia. Se denomina *arista de separación* a aquella arista de un grafo conexo que, al eliminarla, lo transforma en disconexo.

Aludiendo una vez más a nuestro ejemplo, Figura 3.1, comprobamos que dicho grafo no era conexo puesto que contiene dos clases de equivalencia: $[1] = \{1,2,3\}$ y $[4] = \{4\}$.

Definición 11. Decimos que un grafo $G = (V,E)$ es *regular* de orden n o n-regular si $\text{gr}(v) = n$ para todo $v \in V$, es decir, si todos los vértices tienen el mismo grado.

En la Figura 3.7 encontramos un grafo 3-regular o regular de orden 3.

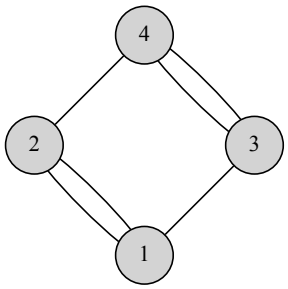


Figura 3.7. Grafo regular de orden 3.

Definición 12. Sea $G = (V,E)$ un grafo. Decimos que G es un *grafo completo* si es un grafo simple y, para todo $u, v \in V,\ \exists\, e \in E, \Phi_G(e) = \{u,v\}$. Es decir, si dados dos vértices cualesquiera, existe una arista que los conecta o, dicho de otra forma, todos los vértices son adyacentes.

Otra caracterización de los grafos completos la podríamos dar a partir del grado de sus vértices, como enunciamos a continuación.

Sea $G = (V,E)$ un grafo simple de orden n. Entonces G es un grafo completo si verifica que $\text{gr}(v) = n-1$ para todo $v \in V$.

Este tipo de grafos se representan por $\mathbb{K}_n$ siendo n el número de vértices. Además, su representación gráfica suele hacerse como un polígono regular. El número de aristas de $\mathbb{K}_n$ es $\binom{n}{2} = \frac{n(n-1)}{2}$. En la Figura 3.8 tenemos una representación del grafo completo $\mathbb{K}_5$ que, efectivamente, tiene 10 aristas.

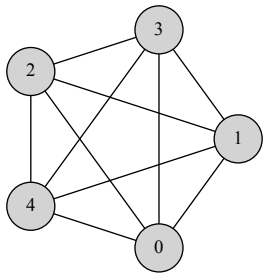


Figura 3.8. Grafo completo de 5 vértices, $\mathbb{K}_5$.

3.2. Isomorfismos de grafos

A la hora de representar o dibujar un grafo podemos hacerlo de distintas formas, las cuales permitirán dilucidar más fácilmente unas características o propiedades que otras. En ocasiones, dos representaciones distintas del mismo grafo pueden hacernos ver grafos distintos, por ello es importante saber abstraer, separar, la estructura del grafo de su dibujo, de su etiquetado. Es aquí donde nace el concepto de isomorfismo de grafos.

Definición 13. Sean los grafos $G = (V, E)$ y $G' = (V', E')$ junto con sus respectivas aplicaciones de incidencia Φ_G y $\Phi_{G'}$. Decimos que los grafos G y G' son *isomorfos* si existen dos biyecciones $h_V : V \to V'$ y $h_E : E \to E'$ tales que para todo $e \in E$ si $\Phi_G(e) = \{u, v\}$, entonces se verifica que $\Phi_{G'}(h_E(e)) = \{h_V(u), h_V(v)\}$. En tal caso diremos que las aplicaciones biyectivas h_V, h_E forman un isomorfismo de G a G'.

Proposición 5. *La relación de isomorfía es una relación de equivalencia.*

- Reflexividad: Todo grafo es isomorfo a sí mismo y las aplicaciones biyectivas que definen tal isomorfismo son la identidad en el conjunto de vértices y el conjunto de aristas, $I_V : V \to V$ y $I_E : E \to E$.
- Simetría: Si hay un isomorfismo del grafo $G = (V, E)$ al grafo $G' = (V', E')$ definido por las aplicaciones biyectivas $h_V : V \to V'$ y $h_E : E \to E'$, entonces las aplicaciones biyectivas ${h_V}^{-1}$ y ${h_E}^{-1}$ forman un isomorfismo de G' a G.
- Transitividad: Sean $F = (V_F, E_F), G = (V_G, E_G)$ y $H = (V_H, E_H)$ tres grafos tales que hay un isomorfismo de F a G definido por las aplicaciones biyectivas $h_{V_F} : V_F \to V_G$ y $h_{E_F} : E_F \to E_G$ y un isomorfismo de G a H definido por las aplicaciones biyectivas $h_{V_G} : V_G \to V_H$ y $h_{E_G} : E_G \to E_H$. Entonces existe un isomorfismo de F a H que viene definido por las aplicaciones biyectivas $h_{V_G} \circ h_{V_F}$ y $h_{E_G} \circ h_{E_F}$.

Proposición 6. *Dos grafos son isomorfos si, y sólo si, lo son sus complementarios.*

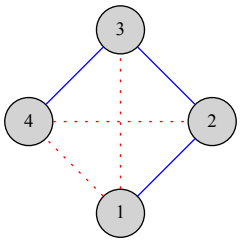


Figura 3.9. Grafos autocomplementarios.

Definición 14. Un grafo G se denomina *autocomplementario* si verifica que G es isomorfo a $\overline{G}$, siendo $\overline{G}$ el complementario de G.

Como vemos en la Figura 3.9, el grafo cuyas aristas están rellenas en color azul es complementario al grafo cuyos lados son lineas de puntos rojas pero, además, tales grafos son también isomorfos.

Detectar cuándo dos grafos son o no isomorfos no suele ser una tarea fácil, por ello, intentamos extraer de ellos datos que necesariamente sean comunes por pertenecer a la misma clase de isomorfía.

Definición 15. Una propiedad se dice *invariante por isomorfismo* si, dados dos grafos isomorfos G y G', uno satisface la propiedad si y solo si lo hace también el otro.

Algunos de estos invariantes son el orden y el tamaño de un grafo o la sucesión de grados de éste. Los invariantes constituyen una condición necesaria para el isomorfismo de grafos, sin embargo, no es suficiente.
Estudiemos ahora si los grafos de la Figura 3.10 son o no isomorfos.

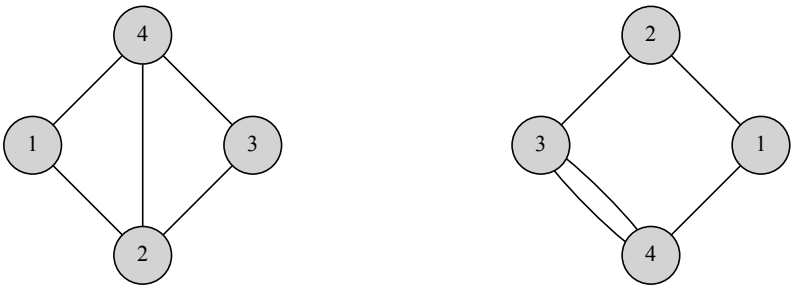


Figura 3.10

Podríamos pensar que ambos grafos son isomorfos, puesto que coinciden en el número vértices (4), en el número de aristas (5), en sus sucesiones de grados

(0 0 2 2) e incluso podemos ver que, en ambos grafos, los dos vértices de grado tres son adyacentes. Sin embargo, no se puede establecer una biyección entre sus conjuntos de vértices y aristas de tales grafos y, por tanto, definir un isomorfismo, dado que en el grafo de la izquierda los vértices de grado dos no son adyacentes, mientras que en el grafo de la derecha sí.

Análogamente, podemos definir el isomorfismo entre grafos dirigidos, atendiendo simplemente a algunos detalles.

Definición 16. Sean $D = (V, E)$ y $D' = (V', E')$ dos digrafos. Decimos que D y D' son *isomorfos* si existe una aplicación biyectiva $h_V : V \to V'$ tal que si la arista dirigida $(u, v) \in E$, entonces $(h_V(u), h_V(v)) \in E'$. Es decir, h_V mantiene el sentido de las aristas de D.

En este caso, el grado entrante y saliente, son invariantes por isomorfismo entre digrafos.

3.3. Matrices y grafos

Como hemos comentado, los grafos pueden ser representados de distintas formas, sin embargo, no todas las representaciones tienen por qué ser gráficas. En esta sección, veremos cómo los grafos pueden ser presentados como matrices, las cuáles, en ocasiones, pueden ser útiles para encontrar con mayor facilidad algunos elementos suyos u obtener ciertas propiedades sobre ellos. Existen varios tipos de matrices asociadas a grafos, entre los que destacan la *matriz de adyacencia* y la *matriz de incidencia*. En esta sección vamos a definir ambas matrices, estudiaremos algunas propiedades que nos brindan sobre los grafos que representan y caracterizaremos algunos de los resultados ya expuestos en este trabajo a través de ellas.

Definición 17. Sea $G = (V, E)$ un grafo de orden n. Definimos su *matriz de adyacencia* como la matriz $A \in M_n(\mathbb{N})$ donde sus elementos a_{ij} corresponden al número de aristas que conectan los vértices $i, j \in V$.

La matriz de adyacencia asociada al grafo de la Figura 3.1 es:

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 0 & 0 \\ 2 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

Podemos enunciar entonces una serie de resultados que nos aportan algunas características y propiedades sobre las matrices de adyacencia.

Proposición 7. *Un grafo queda determinado por su matriz de adyacencia, A. Si se cambian las etiquetas de los vértices, la matriz de adyacencia resultante, C, es distinta, pero existe una matriz permutación, P, que verifica $P^{-1}CP = A$. Además, cualquier matriz $A \in M_n(\mathbb{N})$ se corresponde con la matriz de adyacencia de un grafo finito.*

Nota 1. Recuérdese que una matriz permutación es una matriz obtenida al intercambiar las filas o las columnas de la identidad.

Proposición 8. *La matriz de adyacencia de un grafo es simétrica.*

Proposición 9. *Sea G un grafo de orden n y sea A su matriz de adyacencia. Entonces los elementos a'_{ij} de la matriz A^n determinan el número de caminos de longitud n que unen el vértice i con el vértice j.*

Demostración. Realicemos la demostración por inducción sobre el exponente de la matriz de adyacencia. Si $n = 1$, obtenemos A, que es la matriz de adyacencia y verifica la proposición por cómo es definida. Supongamos ahora el resultado cierto para $n-1$. Definimos $B = A^{n-1}$ y $C = A^n$. Nuestro objetivo entonces es demostrar que los elementos c_{ij} son el número de caminos de longitud n que conectan los vértices $i, j \in V$. Sabemos que $c_{ij} = \sum_{k=1}^{n} b_{ik}a_{kj}$. Dado un camino de longitud $n-1$ entre el vértice i y un vértice cualquiera v, obtendremos un camino de longitud n entre i y j añadiendo la arista que conecta v y j. Por tanto, el número de caminos de longitud n entre i y j se obtiene contando el número de caminos de longitud $n-1$ entre i y $k : k = 1, 2, \ldots, n$ y, para cada uno de ellos, contar el número de aristas que llegan desde k hasta j. Por la hipótesis de inducción esto equivale a:

$$b_{i1}a_{1j} + b_{21}a_{2j} + \ldots + b_{n1}a_{nj} = c_{ij}.$$

Como queríamos demostrar. □

Nota 2. Esta demostración es válida para matrices de adyacencia de grafos dirigidos.

Observando la matriz de adyacencia de un grafo se pueden apreciar algunas características suyas. Como la matriz de adyacencia es simétrica, comentaremos dichas características en base a las filas de la matriz, sabiendo que pueden también aplicarse las siguientes conclusiones fijándonos en las columnas.

Sea entonces (a_{ij}) la matriz de adyacencia de un grafo G de orden n.

- Si el elemento a_{ij} es distinto de cero, significa que los vértices i y j son adyacentes.
- Si el elemento a_{ii} de la diagonal es $b \neq 0$, significa que el vértice i tiene b lazos.
- Si la fila i-ésima es nula, significa que el vértice i es un vértice aislado.
- El grado de un vértice i viene dado como: $\text{gr}(i) = 2a_{ii} + \sum_{i=1, j\neq i}^{n} a_{ij}$.
- Si el elemento $a_{ij} > 1$, entonces existen lados paralelos entre los vértices i y j.

Caractericemos ahora algunos de los resultados y definiciones mencionados en esta sección para un grafo a partir de su matriz de adyacencia.

- Sea D un digrafo con aplicaciones dominio y codominio, O_D y F_D, respectivamente. Entonces el elemento a_{ij} de su matriz de adyacencia determina el número de aristas que tienen como origen el vértice i y como final el vértice j. Las matrices de adyacencia de los digrafos no tienen por qué ser simétricas.

- Un grafo completo $\mathbb{K}_n$ es aquel cuyos elementos de la matriz de adyacencia son cero en la diagonal principal y $n-1$ fuera de ella.

- Sea G un grafo de orden n cuya matriz de adyacencia es A. Si todos los elementos de A fuera de la diagonal son no nulos, entonces G es conexo.

- Dos grafos son isomorfos si se pueden reetiquetar los vértices de uno de tal forma que las matrices de adyacencia de ambos sean iguales. Por tanto, el problema se reduce a encontrar una matriz de permutación que, al aplicarla sobre la matriz de adyacencia de uno de ellos, dé como resultado la matriz de adyacencia del otro.

Definición 18. Sea $G = (V, E)$ un grafo tal que $V = \{v_1, \ldots, v_n\}$, $E = \{e_1, \ldots, e_m\}$. Llamamos *matriz de incidencia* de G a la matriz B de dimensiones $n \times m$ tal que $b_{ij} = 1$ si $v_i \in \Phi_G(e_j)$ y $b_{ij} = 0$ en caso contrario.

La matriz de incidencia asociada al grafo de la Figura 3.1 es:

$$\mathbf{B} = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

Proposición 10. *Si reetiquetamos los vértices o aristas de un grafo, la matriz de incidencia cambia, pero se puede llegar de una a otra realizando operaciones elementales de intercambio de sus filas o columnas.*

Veamos, como en el caso de la matriz de adyacencia, algunas características que deducimos de un grafo según su matriz de incidencia.

- La matriz de incidencia de un grafo no tiene por qué ser simétrica.

- Si en una matriz de incidencia encontramos una fila cuyos elementos son todos nulos, entonces el vértice que representa a esa fila es aislado.

- Una columna de una matriz de incidencia nunca puede ser nula, ya que significaría que esa arista no conecta con ningún vértice, por tanto, no existiría.

- El grado de un vértice viene dado por la suma de todos los elementos de la fila que lo representa pero multiplicando por dos aquellos elementos que sean los únicos valores no nulos en su columna.

- Un grafo posee un lazo cuando su matriz de incidencia tiene una columna con todos sus elementos nulos salvo uno.
- Un grafo tiene aristas paralelas si su matriz de incidencia posee columnas iguales.
- La matriz de adyacencia de un grafo es cuadrada porque relaciona vértices con vértices, mientras que la matriz de incidencia no tiene por qué serlo, ya que relaciona vértices y aristas.

Análogamente a lo realizado con la matriz de adyacencia, caractericemos algunos resultados sobre grafos en función de la matriz de incidencia.

- Sea D un digrafo. La matriz de incidencia de D toma el valor -1 para el vértice origen y el valor 1 para el vértice final. Notemos que no se puede definir la matriz de incidencia de un digrafo con lazos.
- Una matriz de incidencia corresponde a un grafo completo si cumple las siguientes dos condiciones:
  i. Tiene n filas si, y sólo si, tiene $\frac{n(n-1)}{2}$ columnas,
  ii. En cada fila tiene $n-1$ elementos igual a 1 y el resto son ceros.

Capítulo 4

Grafos de Euler y grafos de Hamilton

4.1. Grafos Eulerianos

Definición 19. Sea G un grafo conexo, se dice que G tiene un *camino de Euler* si existe en G un recorrido en el que aparezcan todas las aristas. Diremos que G tiene un *circuito de Euler* si el camino de Euler es también un camino cerrado. Se define un *grafo de Euler* (o *grafo Euleriano*) como aquel que contiene un circuito de Euler.

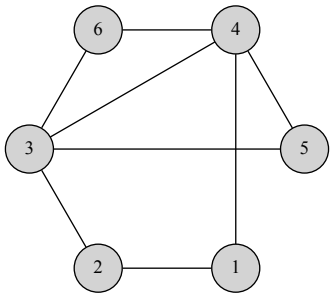


Figura 4.1. Grafo de Euler.

El grafo de la Figura 4.1 es un grafo Euleriano, dado que posee un circuito de Euler, el determinado por los vértices $1-2-3-4-5-3-6-4-1$.

El grafo de la Figura 4.2 no es un grafo Euleriano, pero posee un camino de Euler, determinado por los vértices $3-2-1-5-3-4-2-5-4$.

En los ejemplos anteriores podemos comprobar cómo, en el caso del grafo Euleriano, todos los vértices tienen grado 2 ó 4, mientras que en el otro caso observamos como dos vértices tienen grado 4, un vértice tiene grado 2 y, los vértices inicial y final del camino, tienen ambos grado 3.

Teorema 1. *Sea G un grafo conexo. Entonces G es un grafo de Euler si, y sólo si, el grado de cada vértice es par.*

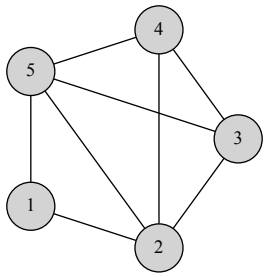


Figura 4.2. Grafo con un camino de Euler.

Demostración. Necesidad: Supongamos que G tiene un circuito de Euler, entonces recorre todos los vértices, por tanto, G es conexo. Además, al recorrer el circuito, entramos y salimos de un vértice por aristas distintas, por tanto, el grado de cada vértice es par.

Suficiencia: Realizaremos esta demostración mediante inducción sobre el número de aristas, n. Sea $n = 0$, al ser el grafo conexo, concluimos que sólo tiene un nodo, de grado 0, par. Sea $n = 1$, si el grafo es conexo y cada vértice debe tener grado par, la única posibilidad es un grafo de un sólo vértice, en cuyo caso, la única arista existente es un lazo y, por tanto, ya tendríamos el circuito de Euler buscado.

Supongamos cierto el resultado para cualquier grafo $G = (V, E)$ conexo de tamaño menor que n, $n \in \mathbb{N}$ y tal que $\text{gr}(v)$ es par para todo $v \in V$.

Sea ahora G un grafo conexo con n aristas y grado par en todos sus vértices. Al ser $\text{gr}(v)$ par y ser conexo, no tiene puntos aislados, en particular $\text{gr}(v) > 1$, así por el Lema 1 (Sección 3.1) sabemos que existe un circuito en G, sea éste C. Si eliminamos de G este circuito, se sigue teniendo que $\text{gr}(v)$ es par, para todo $v \in V$, ya que, en cada vértice, hemos eliminado la arista por la que llegamos hacia él y por la que partimos desde él. Es posible que el grafo resultante no sea conexo pero, para cada componente conexa que tenga al menos una arista, tendremos un circuito de Euler, pues hemos supuesto cierto el resultado para todo grafo de tamaño menor que n, llamemos a estos circuitos $c_1, c_2 \ldots, c_m$. Además, en cada componente conexa (y por tanto, en cada circuito c_i), hay un vértice, v_i, que interviene en C.

Así pues, recorremos C de tal forma que, al llegar a v_i, insertemos el circuito c_i. Al terminar, tendremos un nuevo circuito que incluirá todas las aristas de G recorriéndolas una sola vez, por tanto, dicho circuito será de Euler, como queríamos demostrar.

□

Corolario 1. *Sea G un grafo conexo, entonces G tiene un camino de Euler si, y sólo si, tiene exactamente dos vértices de grado impar.*

Demostración. Supongamos que u y v son los dos vértices de grado impar. Añadimos en G la arista (u, v), por tanto, obtenemos ya un grafo conexo cuyos vértices tienen todos grado par. Por el Teorema 1 sabemos que en G existe un circuito de Euler, en particular, podemos suponer que dicho circuito comienza en el vértice u o en el vértice v. Al suprimir de este circuito la arista (u, v), obtendremos entonces un camino entre u y v que recorre las aristas una sola vez, es decir, obtendremos el camino de Euler que buscábamos. □

Así pues, nos encontramos ya en condiciones de resolver el famoso problema con el que comenzó la Teoría de Grafos, el problema de los puentes de Königsberg. Volviendo a la Figura 2.2, podemos ver que el grafo que la representa contiene tres vértices de grado impar, entonces, por teorema anterior, podemos concluir que no existe un circuito de Euler, es decir, no pueden recorrerse todos los puentes una sola vez empezando y acabando en el mismo punto.

Teorema 2. *Sea G un grafo conexo. Entonces G es Euleriano si, y sólo si, admite una descomposición en ciclos.*

Nota 3. Decimos que un grafo admite una descomposición en ciclos cuando todas sus aristas quedan repartidas en conjuntos disjuntos, donde cada conjunto representa un ciclo en el grafo (recorridos sus elementos de manera pertinente). Sea $c_1, c_2, \ldots, c_n$ la descomposición en ciclos de un grafo G, entonces escribiremos $G = c_1 \oplus c_2 \oplus \cdots \oplus c_n$.

> *Demostración. Suficiencia*: Consideramos en G los ciclos disjuntos en aristas $c_1, \ldots, c_m$ tal que $G = c_1 \oplus \ldots \oplus c_m$. Para cada ciclo $c_1, \ldots, c_m$, todos los vértices tienen grado dos, por tanto, al ser G conexo, cada vértice de G tendrá grado par.
>
> *Necesidad*: Supongamos que G es un grafo de Euler y llamemos C a un circuito de Euler en G. Si en C no aparece repetido ningún vértice, entonces C es un ciclo. En caso contrario, llamemos v_1 a este primer vértice repetido y sea c_1 el ciclo que empieza y acaba en v_1. Eliminamos c_1 de G y nos queda un grafo conexo salvo vértices aislados con todos sus vértices de grado par. Entonces el grafo resultante es Euleriano. Repetimos este procedimiento hasta haber recorrido todas las aristas de G, en tal caso, tendremos un conjunto de ciclos, disjuntos en aristas, cuya unión es G.
>
> □

Gracias a estos resultados, podemos saber si un grafo tiene caminos o circuitos de Euler, de hecho, nos dan métodos para construirlos.

La utilidad de este tipo de grafos se hace visible puesto que permite recorrer todas las aristas del grafo. Por ello, son muy aplicables en situaciones reales como la construcción de redes de carreteras, permitiendo incluso volver al mismo punto de partida (si hablamos de grafos Eulerianos).

Pues bien, presentamos a continuación dos algoritmos que nos permiten construir los caminos o circuitos de Euler.

A.1 Algoritmo de Fleury

Supongamos G un grafo no trivial.

1. Comprobamos que el grafo es conexo y que todos los vértices tienen grado par. En caso de haber vértices con grado impar, se comprueba que sólo sean dos.

2. Creamos una lista $C = []$ donde iremos introduciendo los vértices o aristas que determinan el camino o circuito Euleriano. Si el grafo es Euleriano, empezamos el camino en cualquiera de sus vértices, en caso contrario, escogemos uno de los dos vértices con grado impar. Llamemos v a este vértice de partida.

3. Si v tiene más de una arista adyacente, escogemos una arista que no sea de separación (en caso de tener sólo una arista adyacente, la escogemos), llamamos w a su otro extremo, incluimos (v, w) en C, borramos dicha arista del grafo y nos movemos de v a w. Si al eliminar la arista obtenemos un vértice aislado, lo eliminamos también.

4. Si el grafo no es trivial, volvemos a 2, en caso contrario, devolvemos C y terminamos el proceso. La sucesión de vértices o aristas de C es el circuito o camino Euleriano.

Ejemplifiquemos este algoritmo observando cómo funciona al aplicarlo sobre el grafo de la Figura 4.3, al que vamos a referirnos como G.

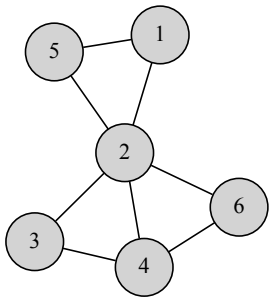
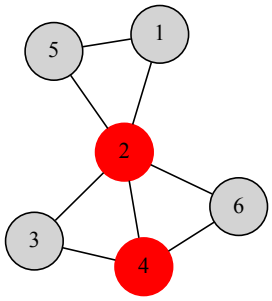


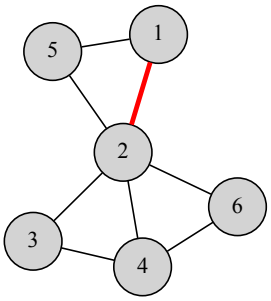
Figura 4.3

En primer lugar, debemos determinar si en el grafo estudiado existe un camino o un circuito de Euler. En nuestro caso, podemos ver que todos los vértices tienen grado par, excepto los nodos 2 y 4, que tienen grados 5 y 3, respectivamente. Por tanto, en virtud del *Corolario 1* de esta misma sección, podemos afirmar que posee un *Camino de Euler*.

Así pues, como indica el algoritmo, el camino empezará en uno de los vértices de grado impar.

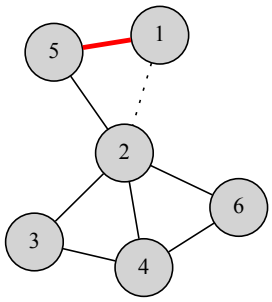


Escojamos como vértice inicial, por ejemplo, el vértice 2. Siguiendo las instrucciones del método, debemos ahora escoger una arista incidente en 2 que no sea de separación, es decir, que no haga el grafo disconexo. Sea ésta, por ejemplo, la arista $(2,1)$. Añadimos entonces ese lado a nuestra lista C, donde quedará definido el camino de Euler.



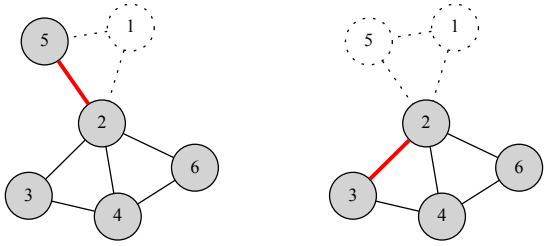
$$C = [(2,1)]$$

Nos posicionamos entonces en el extremo opuesto de la arista, el vértice 1, eliminamos la arista $(2,1)$ y, de nuevo, buscamos una arista incidente en 1 que no sea de separación. Como sólo la arista $(1,5)$ incide en el nodo 1, la escogemos y la añadimos a C.

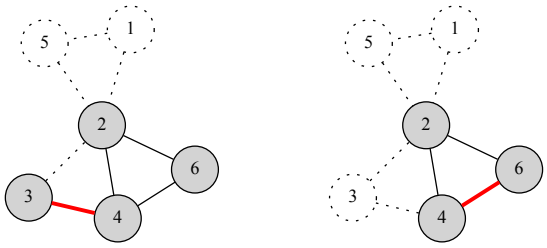


$$C = [(2,1),(1,5)]$$

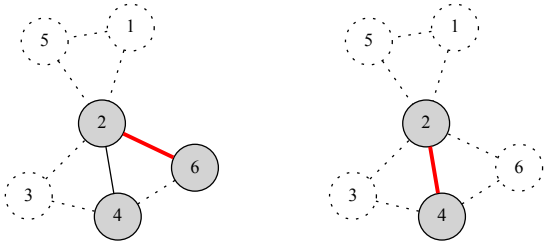
De nuevo, nos posicionamos ahora en el vértice opuesto de la arista $(1,5)$, es decir, en el nodo 5. Eliminamos dicha arista y, al quedar aislado el nodo 1, lo eliminamos también. Repetimos entonces el proceso hasta que hayamos recorrido todas las aristas de G.



$$C = [(2,1),(1,5),(5,2),(2,3)]$$



$$C = [(2,1),(1,5),(5,2),(2,3),(3,4),(4,6)]$$



$$C = [(2,1),(1,5),(5,2),(2,3),(3,4),(4,6),(6,2),(2,4)]$$

De esta forma, C determina un camino de Euler en el grafo G.

En nuestro repositorio [7] podemos encontrar una implementación del algoritmo de Fleury. A modo de ejemplos, lo hemos aplicado sobre el grafo de la Figura 4.7.

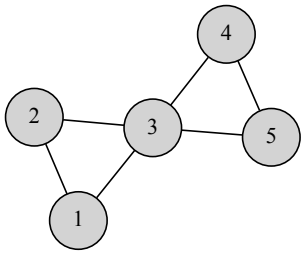


Figura 4.7

El código para su ejecución es el siguiente:

```
from Fleury import *

g=Grafo([],[(1,2),(1,3),(2,3),(3,4),(3,5),(4,5)])

Fleury(g,True)
```

El elemento `Grafo()`, admite como argumentos dos listas, la primera con los vértices que contendrá y la segunda, con las aristas (si la primera es vacía, asumirá que los vértices son todos los que aparecen en la especificación de las aristas). Además, todos los algoritmos implementados requieren como primer argumento un grafo y, como segundo, opcional, la sentencia lógica `True` si se desea obtener la salida de la función paso a paso o, si no se introduce nada, se retornará de la función el grafo final resultante.

Y la salida por pantalla se corresponde con la Figura 4.8.

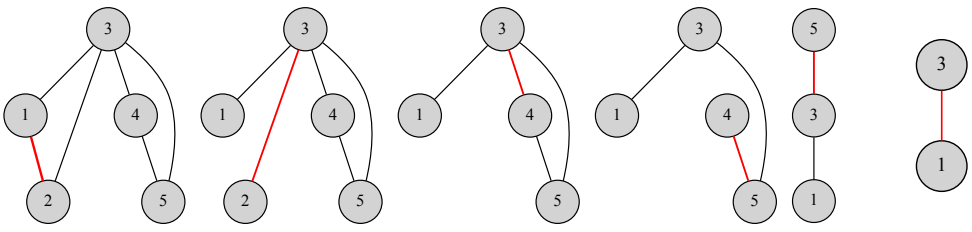


Figura 4.8. Algoritmo de Fleury.

El algoritmo de Fleury es sencillo de entender, sin embargo, en ocasiones es difícil determinar si al eliminar la arista escogida, el grafo sigue siendo conexo (exceptuando los nodos aislados). Una alternativa a éste es el algoritmo de Hierholzer [4], más sencillo de programar y con el mismo coste computacional.

A.2 Algoritmo de Hierholzer

1. Comprobamos que el grafo es conexo y que todos los vértices tienen grado par. En caso de haber vértices con grado impar, se comprueba que sólo sean dos.

2. Creamos una lista $C = []$ donde iremos introduciendo los vértices que determinan el camino o circuito Euleriano. Si el grafo es Euleriano, empezamos el camino en cualquiera de sus vértices, en caso contrario, escogemos uno de los dos vértices con grado impar, además añadimos una arista uniendo los dos vértices de grado impar. Llamemos v a este vértice de partida y lo incluimos en C.

3. Buscamos un ciclo en v y vamos añadiendo los vértices que lo determinan a C. Eliminamos las aristas involucradas en dicho ciclo y, si surgiesen, los vértices aislados.

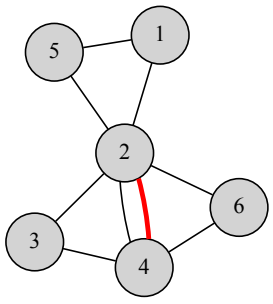
4. Si el ciclo formado es un circuito de Euler:

   - Si el grafo G era Euleriano, devolvemos C y terminamos el proceso.
   - Si sólo contenía un camino de Euler, eliminamos de C la aparición del vértice inicialmente escogido que sea adyacente al otro vértice de grado impar.

   En caso contrario, dentro del ciclo formado, buscamos un vértice w que tenga aún aristas adyacentes no consideradas en el ciclo anterior, buscamos un ciclo para este vértice y sustituimos la sucesión de vértices que lo determinan en el lugar que ocupa w en C.

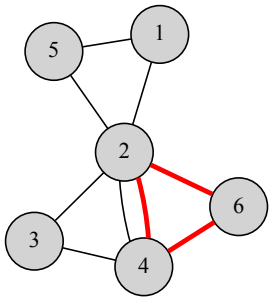
Veamos como funciona este algoritmo sobre el grafo de la Figura 4.3, que sabemos que tiene un camino de Euler.

El primer paso es incorporar una arista que conecte los dos vértices de grado impar de G, es decir, los nodos 2 y 4. Así pues, nuestro grafo de partida es el siguiente, donde escogemos, por ejemplo, el vértice 2 como vértice inicial.

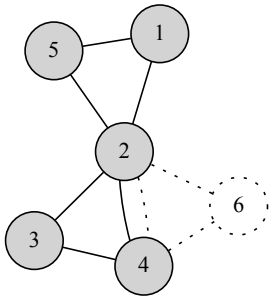


$$C = [2]$$

El siguiente paso es buscar un ciclo en 2 y añadir los vértices que lo determinan a C. Un ciclo podría ser, por ejemplo, el formado por las aristas marcadas en rojo:

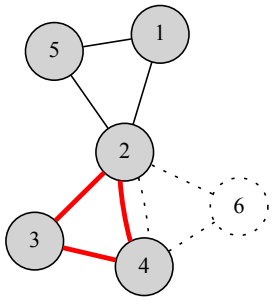


Añadimos entonces los vértices que determinan el ciclo a C y borramos las aristas utilizadas y los vértices que, en consecuencia, queden aislados, como el vértice 6.

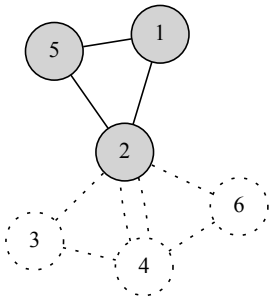


$$C = [2, 4, 6, 2]$$

Escogemos ahora un vértice de C que no haya sido borrado (esto significa que aún tiene aristas incidentes) y, de nuevo, buscamos un ciclo sobre él. Elijamos el vértice 4 y el ciclo $4-3-2-4$, por ejemplo.



Añadimos los vértices que determinan ese nuevo ciclo a una lista distinta a C y borramos las aristas utilizadas y los vértices aislados.



$$C = [2,4,6,2],\ C_1 = [4,3,2,4]$$

Repetiríamos este proceso hasta obtener la partición por ciclos disjuntos en aristas de G. En nuestro caso, solo nos queda el ciclo C_2=[2,1,5,2]. Empezamos el circuito en el vértice 2 porque el algoritmo indica que el vértice inicial de un nuevo ciclo debe ser uno presente en C.

Así pues, tenemos ya la partición en ciclos de G:

$$C = [2,4,6,2],\ \ C_1 = [4,3,2,4] \text{ y } C_2 = [2,1,5,2]$$

Solo nos queda incluir todos los ciclos obtenidos en nuestro ciclo inicial C:

- $C \oplus C_1 = [2,4,3,2,4,6,2]$
- $C \oplus C_1 \oplus C_2 = [2,4,3,2,4,6,2,1,5,2]$

Finalmente, debemos retirar la arista que incluimos al principio del proceso por ser un camino de Euler lo que existía en G. Así, eliminamos de C el primer 2 que nos encontramos, puesto que precede al vértice 4, es decir, determinan la arista $(2,4)$ que habíamos añadido al inicio del algoritmo. Tenemos entonces que un camino de Euler en el grafo G viene dado por los vértices $C = [4,3,2,4,6,2,1,5,2]$.

El algoritmo de Hierholzer [4] está basado fundamentalmente en un método creado para nuestra clase *Grafo* que nos devuelve los distintos ciclos que podemos encontrar en él. Este método consiste en buscar, dado un vértice u sobre el que se van a buscar los ciclos, los caminos simples que hay desde sus nodos vecinos $v_1, v_2, \ldots, v_n$ hasta él. Así, obtenemos caminos simples de la forma $[v_i, \ldots, u]$, $i \in \{1,2,\ldots,n\}$. Por tanto, como v_i es vecino de u, podemos añadir a ese camino la arista (u, v_i) y obtenemos el ciclo $[u, v_i, \ldots, u]$, $i \in \{1,2,\ldots,n\}$.

Una implementación de este algoritmo la encontramos en nuestro repositorio [7]. A modo de ejemplo, lo hemos aplicado sobre el grafo de la Figura 4.9.

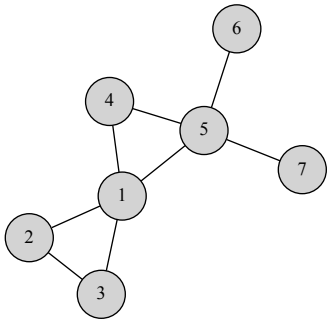


Figura 4.9

El código para su implementación es:

```
from Hierholzer import *

g=Grafo([],[(1,2),(1,3),(1,4),(1,5),(2,3),(4,5),(5,6),(5,7)])

Hierholzer(g,True)
```

Y la salida que obtendremos por pantalla la encontramos en la Figura 4.10.

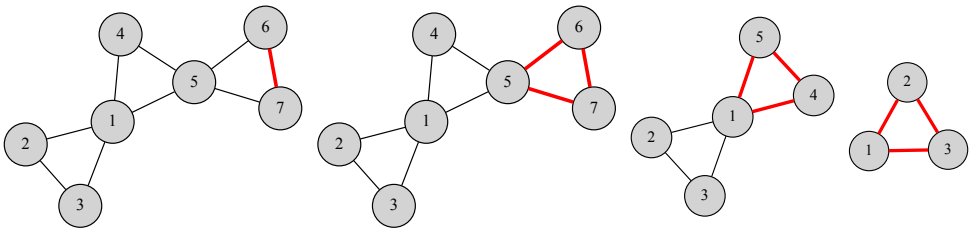


Figura 4.10. Algoritmo de Hierholzer.

4.2. Grafos Hamiltonianos

Definición 20. Sea G un grafo conexo, se dice que G tiene un *camino de Hamilton* si existe en G un camino que recorre todos los vértices una sola vez. Diremos que

G tiene un *circuito de Hamilton* si el camino de Hamilton es también un camino cerrado (en este caso sólo los vértices inicial y final se repetirían). Se define un *grafo Hamiltoniano* como aquel que contiene un circuito de Hamilton.

Figura 4.11. A la izquierda un grafo Hamiltoniano. A la derecha un grafo con un camino de Hamilton.

El primer grafo de la Figura 4.11 es un grafo Hamiltoniano, dado que posee un circuito de Hamilton, definido por los vértices $1-2-3-4-1$, sin embargo, el segundo solo contiene caminos de Hamilton, como por ejemplo los definidos por los vértices $1-2-3-4$ ó $3-4-2-1$, entre otros. Es claro que el segundo grafo no puede ser Hamiltoniano, puesto que en un circuito Hamiltoniano, cada vértice tiene, al menos, dos aristas adyacentes, una para llegar al vértice y otra distinta para partir desde él.

Por otro lado, si en un camino o circuito de Hamilton no podemos pasar dos veces por el mismo vértice, no tiene sentido considerar grafos con aristas paralelas o lazos, por este motivo, en adelante, supondremos que trabajamos con grafos simples.

Teorema 3. *Sea $G=(V,E)$ un grafo conexo con $|V|=n$. Si $|E|\geq \frac{1}{2}(n-1)(n-2)+2$, entonces G es Hamiltoniano.*

La demostración de este teorema la podemos encontrar en [5].

Teorema 4 (Teorema de Dirac)**.** *Sea $G=(V,E)$ un grafo conexo con $|V|=n, n\geq 3$. Si $\mathrm{gr}(v)\geq \frac{n}{2}$ para todo $v\in V$, entonces G es Hamiltoniano.*

Teorema 5 (Teorema de Ore)**.** *Sea $G=(V,E)$ un grafo conexo con $|V|=n, n\geq 3$. Si $\mathrm{gr}(u)+\mathrm{gr}(v)\geq n$ para todo par de vértices $u,v\in V$ no adyacentes, entonces G es Hamiltoniano.*

El Teorema de Dirac se puede deducir del Teorema de Ore pues, si gr(v) $\geq \frac{n}{2}$ para todo $v\in V$, entonces $\mathrm{gr}(u)+\mathrm{gr}(v)\geq n$ para todo par de vértices $u,v\in V$ no adyacentes. Demostremos entonces el Teorema de Ore.

Demostración. Realizamos la demostración por reducción al absurdo. Supongamos un grafo sin circuitos de Hamilton donde $\mathrm{gr}(u) + \mathrm{gr}(v) \geq n$ para todo par de vértices $u, v \in V$ no adyacentes.

Añadimos lados a $G = (V, E)$ hasta obtener un subgrafo $H = (V', E')$ de $\mathbb{K}_n = (V_{\mathbb{K}_n}, E_{\mathbb{K}_n})$ sin circuitos de Hamilton pero que, dada $e \in E_{\mathbb{K}_n}$ con $e \notin E'$, $H + e$ sea un grafo de Hamilton, que es posible construirlo dado que $\mathbb{K}_n$ es Hamiltoniano.

Al ser H subgrafo de $\mathbb{K}_n$, existen $u, v \in V'$ tales que $(u, v) \notin E'$ pero $H + (u, v)$ contiene un circuito de Hamilton C. Como H no contiene a C, podemos deducir que $(u, v) \in C$. Sea $C = [u = v_1, v = v_2, v_3, \ldots, v_n, u]$.

Para todo $i, n \geq i \geq 3$, si el lado (v, v_i) está en H, entonces el lado (u, v_{i-1}) no puede estar en H, ya que, si ambas estuvieran, obtendríamos el ciclo de Hamilton $[v, v_i, v_{i+1}, \ldots, v_{n-1}, v_n, u, v_{i-1}, \ldots, v]$ en H, el cuál, recordemos, no contiene.

En consecuencia, $\mathrm{gr_H}(u) + \mathrm{gr_H}(v) < n$. Por otro lado, para todo $v \in V$, $\mathrm{gr_H}(v) \geq \mathrm{gr_G}(v)$ y, por lo que tenemos dos vértices u y v no adyacentes (en G) que cumplen que $\mathrm{gr_G}(u) + \mathrm{gr_G}(v) < n$, contradiciendo la hipótesis. Por tanto, G es un grafo de Hamilton. □

De nuevo, los resultados recién enunciados sólo hacen referencia a la existencia de caminos o circuitos Hamiltonianos, pero no nos dicen cómo conseguirlos. El algoritmo de Kauffman-Malgrange nos permite encontrar los caminos Hamiltonianos de un grafo cualquiera. Debido a su extensión, lo explicaremos brevemente, en [12] se exponen detalladamente tanto los elementos y definiciones que se requieren, como su funcionamiento.

A.3 Algoritmo de Kauffman-Malgrange

1. El algoritmo comienza creando, para el grafo en cuestión, una matriz M_1 similar a la de adyacencia donde, cuando dos vértices i, j son adyacentes, en lugar de hacer $m_{ij} = a$, siendo a el número de aristas que unen i con j, $m_{ij} = (i, j)$. Es decir, en el elemento m_{ij} de la matriz M, escribimos, si la hay, la arista que conecta esos dos vértices en el grafo. De no haber arista uniendo dichos vértices, se escribirá cualquier símbolo que represente la no existencia, por ejemplo, un guión. En el grafo de la derecha de la Figura 4.11, la matriz inicial M sería:

$$\mathbf{M_1} = \begin{pmatrix} - & (1,2) & - & - \\ (2,1) & - & (2,3) & (2,4) \\ - & (3,2) & - & (3,4) \\ - & (4,2) & (4,3) & - \end{pmatrix}.$$

2. Sea $r = 1$. Realizamos iterativamente la *multiplicación latina de matrices* para obtener $M_{r+1} = M_r * M_1$. Realizamos este procedimiento hasta que $r = n - 1$ siendo n el orden del grafo.

   La multiplicación latina de matrices consiste en proceder como en el producto habitual de matrices, pero entendiéndolo como concatenación de elementos, no como su multiplicación. Recordemos que el primer producto es

$M_1 * M_1$, por tanto, supongamos el producto de los elementos de la fila i, $m_{ij} = (a,b)$ y la columna j, $m_{jk} = (c,d)$ para ejemplificar el procedimiento.

- Si $b = c$, la concatenación de ambas aristas daría como resultado el camino $[a,b,d]$. Colocamos entonces ese camino en el elemento m_{ik} de la matriz M_2.
- Si $b \neq c$, las aristas no se pueden concatenar, en tal caso, colocaríamos un guión $-$ en el elemento m_{ik} de M_2.
- Si $(c,d) = (b,a)$, obtendríamos el circuito $[a,b,a]$, en tal caso, de nuevo, colocamos un guión $-$ en el elemento m_{ik} de M_2.
- Si alguno de los elementos m_{ij} o m_{jk} es un guión, el elemento resultante de su producto también lo será.

Puede ocurrir que, al multiplicar los elementos de la fila i y la columna j, obtengamos varios caminos válidos, en tal caso, escribimos todos ellos en el elemento m_{ik} de la nueva matriz. De esta forma, al realizar de nuevo la multiplicación latina de matrices y multiplicar por el elemento m_{ik}, si éste tiene varios caminos, se multiplicará por cada uno de ellos.

En nuestro caso, el producto $M_2 = M_1 * M_1$ daría como resultado:

$$\mathbf{M_2} = \begin{pmatrix} \begin{pmatrix}1&2&1\end{pmatrix} & - & \begin{pmatrix}1&2&3\end{pmatrix} & \begin{pmatrix}1&2&4\end{pmatrix} \\ - & \begin{pmatrix}2&3&2\\2&1&2\\2&4&2\end{pmatrix} & \begin{pmatrix}2&4&3\end{pmatrix} & \begin{pmatrix}2&3&4\end{pmatrix} \\ \begin{pmatrix}3&2&1\end{pmatrix} & \begin{pmatrix}3&4&2\end{pmatrix} & \begin{pmatrix}3&2&3\\3&4&3\end{pmatrix} & \begin{pmatrix}3&2&4\end{pmatrix} \\ \begin{pmatrix}4&2&1\end{pmatrix} & \begin{pmatrix}4&3&2\end{pmatrix} & \begin{pmatrix}4&2&3\end{pmatrix} & \begin{pmatrix}4&2&4\\4&3&4\end{pmatrix} \end{pmatrix}.$$

Recordemos que debemos eliminar los ciclos de dicha matriz, por tanto, el resultado final sería:

$$\mathbf{M_2} = \begin{pmatrix} - & - & \begin{pmatrix}1&2&3\end{pmatrix} & \begin{pmatrix}1&2&4\end{pmatrix} \\ - & - & \begin{pmatrix}2&4&3\end{pmatrix} & \begin{pmatrix}2&3&4\end{pmatrix} \\ \begin{pmatrix}3&2&1\end{pmatrix} & \begin{pmatrix}3&4&2\end{pmatrix} & - & \begin{pmatrix}3&2&4\end{pmatrix} \\ \begin{pmatrix}4&2&1\end{pmatrix} & \begin{pmatrix}4&3&2\end{pmatrix} & \begin{pmatrix}4&2&3\end{pmatrix} & - \end{pmatrix}.$$

3. Si $r = n-1$, entonces termina el proceso y la matriz M_{n-1} contiene todos los caminos Hamiltonianos del grafo estudiado.

   En nuestro caso, deberíamos llegar hasta $M_3 = M_2 * M_1$, por lo que faltaría

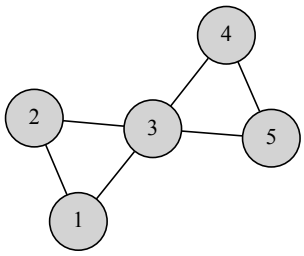


Figura 4.12

realizar una última iteración. El resultado de la misma sería:

$$\mathbf{M_3} = \begin{pmatrix} - & \begin{pmatrix} 1&2&3&2\\1&2&4&2\end{pmatrix} & \begin{pmatrix}1&2&4&3\end{pmatrix} & \begin{pmatrix}1&2&3&4\end{pmatrix} \\ - & \begin{pmatrix} 2&4&3&2\\2&3&4&2\end{pmatrix} & \begin{pmatrix}2&3&4&3\end{pmatrix} & \begin{pmatrix}2&4&3&4\end{pmatrix} \\ \begin{pmatrix}3&4&2&1\end{pmatrix} & \begin{pmatrix} 3&2&1&2\\3&2&4&2\end{pmatrix} & \begin{pmatrix} 3&4&2&3\\3&2&4&3\end{pmatrix} & \begin{pmatrix}3&4&2&4\end{pmatrix} \\ \begin{pmatrix}4&3&2&1\end{pmatrix} & \begin{pmatrix} 4&2&1&2\\4&2&3&2\end{pmatrix} & \begin{pmatrix}4&3&2&3\end{pmatrix} & \begin{pmatrix} 4&2&3&4\\4&3&2&4\end{pmatrix} \end{pmatrix}.$$

Eliminando ahora todos aquellos caminos que contengan ciclos, obtenemos que todos los posibles caminos Hamiltonianos de ese grafo son:

$$\mathbf{M_3} = \begin{pmatrix} - & - & \begin{pmatrix}1&2&4&3\end{pmatrix} & \begin{pmatrix}1&2&3&4\end{pmatrix} \\ - & - & - & - \\ \begin{pmatrix}3&4&2&1\end{pmatrix} & - & - & - \\ \begin{pmatrix}4&3&2&1\end{pmatrix} & - & - & - \end{pmatrix}.$$

La implementación de este algoritmo la encontramos en nuestro repositorio [7]. A modo de ejemplo, lo hemos aplicado sobre el grafo de la Figura 4.12. En ella se realizan 3 pasos:

1. Se realiza la multiplicación latina de matrices.
2. Se realiza una primera limpieza de la matriz resultante donde sustituimos los ciclos por el símbolo ∞.
3. Se realiza una segunda limpieza donde, si en un mismo elemento tenemos varios ∞ (procedentes de varios ciclos), se sustituyen todos por uno solo.

El código a ejecutar es el siguiente:

```
from Kauffman_Malgrange import *

g=Grafo([],[(1,2),(1,3),(2,3),(3,4),(3,5),(4,5)])

Kauffman_Malgrange(g, True)
```

Y la salida por pantalla es:

```
[[inf, (1, 2), (1, 3), inf, inf],
[(2, 1), inf, (2, 3), inf, inf],
[(3, 1), (3, 2), inf, (3, 4), (3, 5)],
[inf, inf, (4, 3), inf, (4, 5)],
[inf, inf, (5, 3), (5, 4), inf]]

[[inf, (1, 3, 2), (1, 2, 3), (1, 3, 4), (1, 3, 5)],
[(2, 3, 1), inf, (2, 1, 3), (2, 3, 4), (2, 3, 5)],
[(3, 2, 1), (3, 1, 2), inf, (3, 5, 4), (3, 4, 5)],
[(4, 3, 1), (4, 3, 2), (4, 5, 3), inf, (4, 3, 5)],
[(5, 3, 1), (5, 3, 2), (5, 4, 3), (5, 3, 4), inf]]

[[inf, inf, inf, [(1, 2, 3, 4), (1, 3, 5, 4)], [(1, 2, 3, 5), (1, 3, 4, 5)]],
[inf, inf, inf, [(2, 1, 3, 4), (2, 3, 5, 4)], [(2, 1, 3, 5), (2, 3, 4, 5)]],
[inf, inf, inf, inf, inf],
[[(4, 3, 2, 1), (4, 5, 3, 1)], [(4, 3, 1, 2), (4, 5, 3, 2)], inf, inf, inf],
[[(5, 3, 2, 1), (5, 4, 3, 1)], [(5, 3, 1, 2), (5, 4, 3, 2)], inf, inf, inf]]

[[inf, inf, inf, (1, 2, 3, 5, 4), (1, 2, 3, 4, 5)],
[inf, inf, inf, (2, 1, 3, 5, 4), (2, 1, 3, 4, 5)],
[inf, inf, inf, inf, inf],
[(4, 5, 3, 2, 1), (4, 5, 3, 1, 2), inf, inf, inf],
[(5, 4, 3, 2, 1), (5, 4, 3, 1, 2), inf, inf, inf]]

Caminos Hamiltonianos:

[(1, 2, 3, 5, 4),
(1, 2, 3, 4, 5),
(2, 1, 3, 5, 4),
(2, 1, 3, 4, 5),
(4, 5, 3, 2, 1),
(4, 5, 3, 1, 2),
(5, 4, 3, 2, 1),
(5, 4, 3, 1, 2)]
```

La búsqueda de caminos o circuitos Hamiltonianos en grafos que representan situaciones reales es muy importante, puesto que nos permite recorrer todos los nodos del grafo una sola vez. Imaginemos por ejemplo un camión de basura, parte de una planta de reciclaje y, tras recorrer todos los contenedores, debe volver a ella, con lo cual, le interesa optimizar el recorrido no pasando dos veces por el mismo punto.

Capítulo 5

Sucesiones gráficas y coloración de grafos

5.1. Sucesiones gráficas

Definición 21. Sea $\{d_k\}_{k\in\{1,\dots,n\}} \subset \mathbb{N}$. Diremos que $\{d_k\}$ es una *sucesión gráfica* si existe un grafo G simple con vértices $\{v_1,\dots,v_n\}$ verificando que $\mathrm{gr}(v_i) = d_i$ para todo $i \in \{1,\dots,n\}$, es decir, si cada elemento de la sucesión coincide con el grado de algún vértice en V. En tal caso, se dice que G es una realización de $\{d_k\}$.

Pongamos algunos ejemplos para aclarar el concepto.

- La sucesión $4,4,4,4,4$ corresponde al grafo completo $\mathbb{K}_5$.

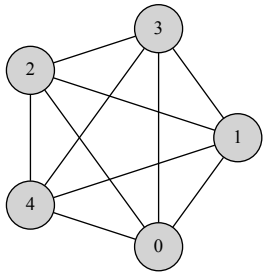


Figura 5.1. Grafo completo $\mathbb{K}_5$.

- La sucesión $3,3,1$ no puede ser gráfica, ya que, un grafo de orden 3 con dos vértices de grado 3 implica que tiene aristas paralelas y, por definición, la realización de una sucesión gráfica ha de ser un grafo simple.
- Sea $\{d_k\}_{k=1,\dots,n}$ tal que $d_1 + d_2 + \dots + d_n = 2m+1$ para algún $m \in \mathbb{N}$. Entonces $\{d_k\}$ nunca puede ser gráfica pues, por *Proposición 2*, sabemos que la suma de los grados de cualquier grafo debe ser par.

Es claro que hemos conseguido algunas condiciones necesarias para que una sucesión de números naturales sea gráfica:

i. La suma de los términos de la sucesión $d_1 + \cdots + d_n$ ha de ser par,

ii. $d_k < n$ para todo $k \in \{1, \ldots, n\}$ siendo n el número de términos de la sucesión, es decir, el orden de su posible realización.

Ahora bien, estas condiciones no son suficientes, observemos el siguiente ejemplo.

Tomemos la sucesión $3, 3, 1, 1$. Esta sucesión debería corresponderse con un grafo G de orden 4. Vemos que la suma de los términos de la sucesión es par y que no hay vértices de orden 4, por lo que no podemos descartar que sea gráfica, es decir, que exista su realización G, siendo G un grafo simple. Sin embargo, tenemos dos vértices de grado 3, es decir, cada uno se une con el resto de los vértices de G, por lo que, en consecuencia, los otros dos vértices restantes van a tener grado 2.

En este ejemplo observamos un método para comprobar que una sucesión no es gráfica, sin embargo, es un poco primitivo y, con un número elevado de vértices, puede ser muy costoso. Por ello, a continuación, enunciamos un resultado que nos permite determinar si una sucesión es o no gráfica, además de aportar los elementos necesarios para, en caso de serlo, construir su realización.

Teorema 6 (Teorema de Havel-Hakimi.)**.** *Sea la sucesión* $\{d_k\}_{k=1,\ldots,n} \subset \mathbb{N}$. *Supongamos que su ordenación de manera no creciente es* $d_1, d_2, \ldots, d_n$ *y que* $d_1 < n$. *Entonces* $\{d_k\}$ *es gráfica si, y sólo si, la sucesión* $\{d_2 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, \ldots, d_n\}$ *también lo es.*

Demostración. Necesidad: Supongamos que existe un grafo $G = (V, E)$, con $V = \{v_1, v_2, \ldots, v_{d_1}, v_{d_1+1}, \ldots, v_{n-1}\}$ cuya sucesión de grados es

$$\{d_2 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, \ldots, d_n\}.$$

Es decir,

$$\mathrm{gr}(v_1) = d_2 - 1,\ \mathrm{gr}(v_2) = d_3 - 1, \ldots,\ \mathrm{gr}(v_{d_1}) = d_{d_1+1} - 1,$$
$$\mathrm{gr}(v_{d_1+1}) = d_{d_1+2}, \ldots, \mathrm{gr}(v_{n-1}) = d_n.$$

Creamos ahora un nuevo grafo G' añadiendo a G el vértice v_0 y las aristas (v_0, v_1), $(v_0, v_2), \ldots, (v_0, v_{d_1})$. Tenemos, por tanto, que $\mathrm{gr}(v_0) = d_1$ y que el grado de los vértices $v_1, v_2, \ldots, v_{d_1}$ se ha incrementado en una unidad.

Así pues, hemos encontrado un grafo cuya sucesión de grados es $\{d_1, d_2, \ldots, d_n\}$

Suficiencia: Supongamos que existe un grafo $G = (V, E)$, con

$$V = \{v_1, v_2, \ldots, v_{d_1}, v_{d_1+1}, \ldots, v_{n-1}\},$$

y tal que $\mathrm{gr}(v_i) = d_i$.

Tenemos entonces que el vértice v_1 es adyacente a d_1 vértices, sean estos, por ejemplo, $v_2, v_3, \ldots, v_{d_1+1}$. Si eliminamos de G el vértice v_1, obtenemos un nuevo grafo $G' = (V', E')$ con $V' = \{v_2, \ldots, v_n\}$ cuya sucesión de grados es $\{d_2 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, \ldots, d_n\}$.

Supongamos ahora que el conjunto de nodos adyacentes a v_1 no es el conjunto de vértices $\{v_2, v_3, \ldots, v_{d_1+1}\}$. En este caso, no podemos proceder igual que en el caso anterior. Lo que haremos será transformar el grafo G en otro grafo con los mismo vértices y el mismo grado para cada vértice pero haciendo que los nodos adyacentes a v_1 sean $v_2, v_3, \ldots, v_{d_1+1}$.

El vértice v_1 no está unido a ningún vértice de $\{v_2, v_3, \ldots, v_{d_1+1}\}$, cojamos entonces un nodo de ese conjunto, por ejemplo, sea éste v_i. Por otro lado, debe haber algún vértice en $\{v_{d_1+2,\ldots,v_n}\}$ adyacente a v_1, sea dicho vértice, por ejemplo, v_j. Por tanto, tenemos dos vértices v_i, v_j tal que la arista (v_1, v_i) no está en el grafo, la arista (v_j, v) sí está y se tiene que $\text{gr}(v_i) \geq \text{gr}(v_j)$.

De la misma forma, existe un vértice v adyacente a v_i pero no a v_j, pues, de no ser así, es decir, si todo vértice adyacente a v_i lo fuera también a v_j, tendríamos que $\text{gr}(v_i) < \text{gr}(v_j)$, que no puede ser.

En resumen, en nuestro grafo G se encuentran los lados (v_1, v_j) y (v_i, v), los cuales eliminamos, mientras que no están las aristas (v_1, v_i) y (v_j, v), las cuáles añadimos. Así, conseguimos un nuevo grafo G' con los mismos vértices de G y cada uno con los mismos grados que los vértices de G pero, además, el vértice v_1 es adyacente, en G', a todos los vértices del conjunto $\{v_2, v_3, \ldots, v_{d_1+1}\}$ que estaban unidos en el grafo G a él y, además, unido también al vértice v_i. Este procedimiento lo repetimos mientras queden nodos en $\{v_2, v_3, \ldots, v_{d_1+1}\}$ que no estén unidos a v_1. Una vez conseguido, procedemos a borrar v_1 y todas las aristas incidentes en él.

□

El teorema de Hawel-Hakimi nos da un algoritmo para saber si una sucesión es gráfica, el cuál explicamos a continuación.

A.4 Algoritmo de demolición

Dada una sucesión $\{d_1, \ldots, d_n\}$ de números naturales.

1. Si todos los elementos de la sucesión son nulos, la sucesión es gráfica. Si la sucesión contiene algún número negativo, no es gráfica.

2. Elimina el mayor elemento, sea éste d_1. A continuación, escoge los d_1 elementos de mayor valor y redúcelos, de uno en uno, por orden de mayor a a menor, en una unidad. (En caso de haber más de d_1 elementos que deberían reducirse, escogemos al azar los elegidos). La sucesión obtenida la denominamos sucesión resultante.

3. Vuelve al paso 1 con la sucesión resultante.

Veamos el funcionamiento de este algoritmo. Para ello, estudiemos si la sucesión $S = \{2,5,2,3,2,2\}$ es una secuencia gráfica.

Eliminamos el término $a = 5$ de S y reducimos en una unidad los 5 mayores términos de $S \setminus \{a\}$. Obtenemos entonces una nueva sucesión $S_1 = \{1,1,2,1,1\}$.

Repetimos ahora el proceso con S_1. Escogemos el término, $b = 2$, y reducimos en una unidad los b términos de mayor valor de $S_1 \setminus \{b\}$. En este caso, los cuatro elementos de $S_1 \setminus \{b\}$ deberían reducirse, pues no hay ninguno mayor que otro, pero el algoritmo dice que sólo debo escoger dos de ellos. Obtenemos así una nueva sucesión $S_2 = \{0,0,1,1\}$.

Finalmente, reiteramos este procedimiento una vez más, sobre S_2, y obtenemos una última sucesión $S_3 = \{0,0,0\}$.

Hemos llegado a una sucesión de ceros, por lo que podemos concluir que la sucesión S es una secuencia gráfica.

Así mismo, otro punto fuerte del Teorema de Havel-Hakimi es que, al determinar si una sucesión es gráfica, vamos generando distintas sucesiones las cuáles nos permiten obtener la realización que buscamos. Es decir, construimos el grafo asociado a la sucesión inicial a partir de las distintas sucesiones resultantes. Exponemos a continuación el algoritmo detalladamente.

A.5 Algoritmo de reconstrucción

Al demoler una sucesión gráfica hemos obtenido consecutivamente una lista R de sucesiones resultantes. Los elementos de cada sucesión de R se corresponden con los grados de los vértices de los sucesivos grafos que formaremos hasta obtener la realización buscada.

1. Partimos de una lista de sucesiones ordenada de menor a mayor en cuanto al número de elementos de dichas sucesiones. Comprobamos que la primera sucesión de la lista es una secuencia de ceros. En otro caso, no existe la realización buscada.

2. Partimos de la sucesión de m ceros, correspondiente a la última sucesión añadida a nuestra lista de entrada. Dibujamos entonces un grafo con m vértices aislados, puesto que la sucesión de m ceros nos indica que hay m nodos de grado cero.

3. Pasamos a la sucesión siguiente añadiendo un vértice al grafo y uniéndolo con tantos vértices como indicara el elemento que se eliminó en la siguiente sucesión para llegar a la actual. Los vértices a los que se une este nuevo vértice son aquellos que incrementan su grado al pasar de la sucesión actual a la siguiente.

4. Si el número de vértices coincide con el número de elementos de la sucesión original, terminamos el proceso.

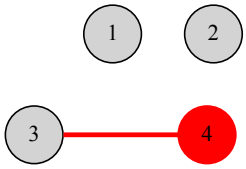
Apliquemos ahora este algoritmo sobre la secuencia gráfica obtenida del algoritmo anterior.

Partimos de la lista de secuencias:

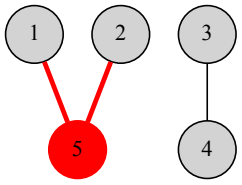
$S_3 = \{0,0,0\}$; $S_2 = \{0,0,1,1\}$; $S_1 = \{1,1,2,1,1\}$; $S = \{2,5,2,3,2,2\}$;

El primer paso es, a partir de la última secuencia obtenida, S_3, construir un grafo con tantos vértices como elementos tenga dicha secuencia y cuyos grados serán los propios elementos de la secuencia. Así, de S_3, obtenemos el siguiente grafo G_1:

El siguiente paso es añadir un vértice a G_1 de tal forma que la sucesión de grados de G_1 sea S_2. Es decir, S_2 me indica que tengo que añadir un vértice 4 y unirlo al vértice 3, que es quién incrementa su grado al pasar de S_3 a S_2. Por tanto, añado la arista $(3,4)$ a G_1.

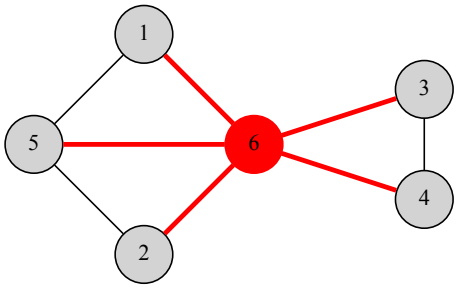


Repetimos el proceso fijándonos ahora en S_1, que nos indica que debemos añadir un vértice 5 y unirlo a los vértices 1 y 2, que dejan de estar aislados para tener grado 1. Así, el nuevo vértice tiene grado 2, como debía ser. Añadamos entonces el vértice 5 y las aristas $(1,5)$ y $(2,5)$. G_1 entonces resultaría tal que así:



Finalmente, fijándonos en S, añadimos un último vértice, el nodo 6, el cuál conectará con los 5 vértices que teníamos, pues observamos que, de S_1 a S, todos los vértices han incrementado en 1 su grado. Por tanto, añado a G_1 las aristas $(1,6),(2,6),(3,6),(4,6)$ y $(5,6)$.

El grafo final, una realización de S, sería la siguiente:



En nuestro repositorio [7] realizamos una implementación de este algoritmo. A modo de ejemplo, lo hemos ejecutado para la lista $[3,2,2,2,1]$, comprobando primero que es secuencia gráfica y, posteriormente, obteniendo su realización.

El código a ejecutar es el siguiente:

```
from grafos import *

g=Grafo()

l=[3,2,2,2,1]

g.Secuencia_a_Grafo(l)
```

Obteniendo como salida la imagen de la Figura 5.2.

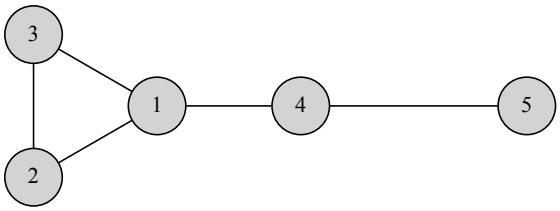


Figura 5.2

En el algoritmo que hemos implementado, el método *Secuencia_a_Grafo* primero comprueba que la lista introducida como argumento sea, en efecto, una sucesión gráfica. En tal caso, construye una realización para ella. De no ser así, muestra el mensaje de error: *La secuencia no es gráfica.*

5.2. Coloración de grafos

La coloración es una operación realizada sobre grafos cuyo objetivo es diferenciar los nodos adyacentes. Pongamos un ejemplo para dejar constancia de su

utilidad. Imaginemos un salón de conferencias con varias salas para impartirse tales actos. Identificamos cada conferencia con un nodo y haremos adyacentes dos nodos cuando las conferencias a los que representan puedan interesar al mismo tipo de público. Se pretende realizar un horario de forma que no se impartan simultáneamente conferencias que puedan resultar interesantes a un mismo grupo de personas. Entonces, asignamos un color a cada franja horaria y coloreamos cada nodo de tal forma que dos conferencias de interés común no tengan el mismo color. Al terminar, cada color marcará una franja horaria y dos conferencias que puedan interesar al mismo tipo de público no serán realizadas a la vez.

Definición 22. Sea $G = (V, E)$ un grafo con aplicación de incidencia Φ_G y C un conjunto (de colores). Una *coloración* de G es una aplicación $f : V \to C$ tal que para todo $e \in E$, si $\Phi_G(e) = \{u, v\}$ tal que $u \neq v$, entonces $f(u) \neq f(v)$.

Es decir, una coloración es una aplicación entre el conjunto de vértices de un grafo y un conjunto de colores de tal forma que dos vértices adyacentes nunca tengan el mismo color.

Definición 23. Sea $G = (V, E)$ un grafo y sean $\{C_1, \ldots, C_n\}, n \in \mathbb{N}$ los distintos conjuntos posibles para los que existe una coloración en G. Se define el *número cromático* de G, y se denota por $\chi(G)$, al $|C_m|$ siendo C_m el conjunto de menor cardinal dentro de $\{C_1, \ldots, C_n\}$. Es decir,

$$\chi(G) = \text{mín}\{|C_1|, \ldots, |C_n|\}$$

Si $\chi(G) = k$, decimos que el grafo es k-coloreable.

En la Figura 5.3 tenemos un ejemplo de un grafo sobre el que se ha aplicado una coloración. Es claro que el número cromático de este grafo es $\chi(G) = 2$.

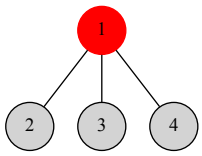


Figura 5.3. Coloración de un grafo.

Así pues, con este nuevo concepto, podemos enunciar algunos resultados para caracterizar definiciones dadas en secciones anteriores:

Proposición 11. *Sea $\mathbb{K}_n$ el grafo completo de orden n. Entonces $\chi(\mathbb{K}_n) = n$.*

Es algo intuitivo dado que todos los vértices son adyacentes entre sí, entonces no puede haber colores repetidos.

Proposición 12. *Sea B un grafo. Entonces B es bipartido si, y sólo si, $\chi(B) = 2$.*

Proposición 13. *Sea G un grafo y sea H un subgrafo de G. Entonces $\chi(H) \leq \chi(G)$.*

Proposición 14. *Un grafo G es trivial si $\chi(G) = 1$. El recíproco no es cierto dado que si $\chi(G) = 1$ significa que G es un grafo vacío (sin aristas), pero no da información sobre el orden de G.*

Proposición 15. *Sea G un grafo y sean $C_1, \ldots, C_n$ sus n componentes conexas, las cuales tienen números cromáticos $\chi(C_1), \ldots, \chi(C_n)$, respectivamente. Se cumple entonces que:*

$$\chi(G) = \text{máx}\{\chi(C_1,) \ldots, \chi(C_n)\}.$$

Proposición 16. *Sea G un grafo cuyo grado máximo es n, entonces:*

i. $\chi(G) \leq n + 1$.

ii. Si G es conexo y no es regular, entonces $\chi(G) \leq n$

Teorema 7. *Sea G un grafo de tamaño m. Entonces:*

$$\chi(G) \leq \frac{1}{2} + \sqrt{\frac{1}{4} + 2m}.$$

En general, no es una tarea fácil conocer el número cromático de un grafo, además, en ocasiones, interesa saber también de cuántas formas distintas se puede colorear un grafo con cierto números de colores, por ello, vamos a definir a continuación una herramienta útil para dichas tareas, el *polinomio cromático.*

Definición 24. Sea G un grafo y sea $x \in \mathbb{N}$. Se define el *polinomio cromático* de G, y se denota por $\text{P}(G, x)$, como el número de coloraciones distintas con x colores que se pueden realizar sobre G.

Pongamos algunos ejemplos para aclarar este concepto:

- Sea G un grafo sin lazos que posee, como mínimo, una arista. Entonces $\text{P}(G, 1) = 0$ dado que G tiene al menos dos vértices adyacentes, por tanto, se necesitan más de un color para realizar la coloración.
- En el caso de los grafos completos, $\mathbb{K}_n$, si disponemos de x colores, para colorear el primer vértice disponemos de los x colores, para colorear el segundo, dado que todos son adyacentes entre sí, solo podemos escoger entonces entre los $x - 1$ restantes. Para el tercer vértice, solo nos quedan $x - 2$ colores donde elegir, y así sucesivamente. De aquí deducimos varios resultados:
  - Si $x < n$, entonces se tiene que $\text{P}(\mathbb{K}_n, x) = 0$, dado que al menos se necesitan n colores para colorear $\mathbb{K}_n$.
  - Si $x = n$, entonces $\text{P}(\mathbb{K}_n, n) = n!$

- Si $x > n$, entonces:

$$P(\mathbb{K}_n, x) = \begin{pmatrix} x \\ n \end{pmatrix} n! = x(x-1)\cdots(x-n+1)$$

Puesto que hay $\begin{pmatrix} x \\ n \end{pmatrix}$ formas de elegir n colores de entre los k posibles y hay $n!$ formas de colorear los vértices con esos n colores.

- Sea G un árbol (véase Capítulo 6) de orden n, entonces su polinomio cromático es: $P(G, x) = x(x-1)^{n-1}$.
- Sea G un grafo de orden n. Si G es un camino simple, entonces su polinomio cromático es $P(G, x) = x(x-1)^{n-1}$.
- Sea G un grafo con m componentes conexas, $G_1, G_2, \ldots, G_m$. Entonces el polinomio cromático de G es $P(G, x) = P(G_1, x)\,P(G_2, x)\cdots P(G_m, x)$.

Teorema 8. *Sea $G = (V, E)$ un grafo simple y sea $e \in E$. Denominemos G_e el grafo G sin la arista e y G'_e al grafo resultante de identificar los vértices adyacentes por e. Se tiene entonces:*

$$P(G_e, x) = P(G, x) + P(G'_e, x).$$

Este último resultado es una herramienta muy útil para calcular el polinomio cromático de un grafo pues, tenemos que, $P(G, x) = P(G_e, x) - P(G'_e, x)$, es decir, podemos calcular el polinomio cromático de un grafo a través de grafos de orden cada vez menor. Pongamos un ejemplo.

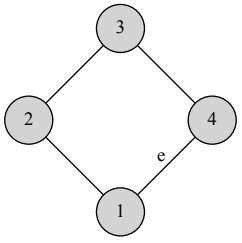


Figura 5.4

En la Figura 5.4 queremos calcular el polinomio cromático de un grafo G 2-regular. Primero, identificamos una arista, en este caso, la arista e que conecta los vértices 1 y 4. El teorema nos dice que el polinomio cromático de nuestro grafo original es igual a la diferencia entre el polinomio cromático de los dos grafos resultantes, uno al quitar e y otro al identificar los vértices adyacentes por e, como vemos en la Figura 5.5 . Así, nos queda que tenemos que calcular el polinomio

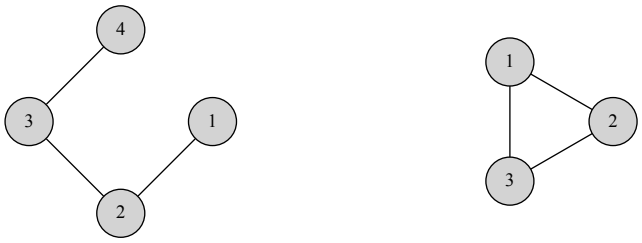


Figura 5.5

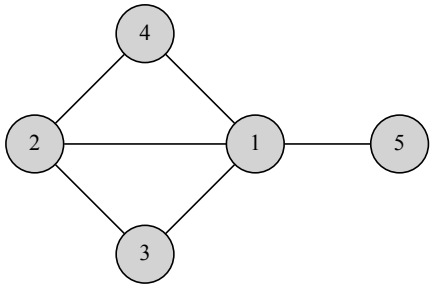


Figura 5.6

cromático de un camino simple S de orden 4 y restarle el polinomio cromático del grafo completo $\mathbb{K}_3$. Entonces tenemos:

$$P(G,x) = P(S,x) - P(\mathbb{K}_3,x) = x(x-1)^{4-1} - x(x-1)(x-2) = x(x-1)(x^2-3x+3).$$

En nuestro repositorio [7] podemos encontrar implementado este algoritmo de dos maneras distintas, de forma iterativa y de forma recursiva. Lo hemos aplicado sobre el grafo de la Figura 5.6.

El código a ejecutar es el siguiente:

```
from Polinomio_Cromatico import *

g=Grafo([],[(1,2),(1,3),(1,4),(1,5),(2,3),(2,4)])

# Forma iterativa

Polinomio_Cromatico_Iterativo(g)

# Forma recursiva

Polinomio_Cromatico_Recursivo(g)
```

La salida obtenida por pantalla es:

```
# Forma iterativa

x**5 - 6*x**4 + 13*x**3 - 12*x**2 + 4*x

# Forma recursiva

x**5 - 6*x**4 + 13*x**3 - 12*x**2 + 4*x
```

Para terminar, cerraremos este apartado exponiendo uno de los resultados más importantes de la coloración de grafos, considerado también como el precursor de esta rama de la Teoría de Grafos, el *Teorema de los cuatro colores*.

El Teorema de los cuatro colores, aunque fue planteado por Francis Guthrie en 1852, fue presentado formalmente por Arthur Cayley en 1878, dice así.

Teorema 9 (Teorema de los cuatro colores.)**.** *Dada cualquier división de un mapa plano en regiones, éste puede ser coloreado únicamente con cuatro colores, de manera que regiones con frontera común tengan colores distintos.*

Al año siguiente de su publicación, el abogado Arthur B. Kempe publicó una demostración del teorema que fue aceptada durante 11 años, hasta que el matemático británico Percy J. Heawood encontró un error, el cual le sirvió para demostrar su *Teorema de los cinco colores*.

Teorema 10 (Teorema de los cinco colores.)**.** *Sea G un grafo plano. Entonces* $\chi(g) \leq 5$.

Finalmente, el teorema de los cuatro colores fue demostrado en 1976 por Appel y Haken y fue el primer gran resultado demostrado con un computador.

Capítulo 6

Árboles

Este capítulo final está dividido en dos objetivos principales: el primero, profundizar en un tipo particular de grafos, los árboles, los cuáles fueron estudiados por primera vez en 1847, por Kirchoff; y que hoy día han adquirido un importante papel en diversos campos de estudio (educación, probabilidad, sistemas digitales, acceso rápido a información, etc.). Hemos caracterizado algunos resultados sobre ellos y hemos implementado diversos algoritmos cuyos objetivos son extraer árboles de un grafo dado. Por otro lado, el segundo objetivo de este capítulo ha sido aportar e implementar métodos que permitieran encontrar un nuevo elemento, de gran importancia, dentro de los grafos: el camino mínimo entre dos vértices. Para ello, hemos intercalado algoritmos que trabajan tanto con la representación gráfica como la representación matricial del grafo.

De esta forma, podemos comenzar este apartado introduciendo algunas definiciones.

Definición 25. Sea T un grafo. Diremos que T es un *árbol* si cumple las siguientes condiciones:

i. T es conexo,

ii. T no contiene ciclos.

Definición 26. Sea F un grafo. Diremos que F es un *bosque* si no contiene ciclos. Es decir, F es un bosque si cada componente conexa $T_1, \ldots, T_n$ es un árbol.

Definición 27. Sea $G = (V, E)$ un grafo y sea $H = (V', E')$ un subgrafo suyo. Diremos que H es un *árbol generador* si cumple que:

i. H es un árbol,

ii. $V' = V$.

En definitiva, cuando hablamos de árboles, estamos tratando con grafos simples, puesto que los lazos y las aristas paralelas son ciclos, por tanto no pueden estar presentes en tales grafos. Además, podríamos definir los árboles como los

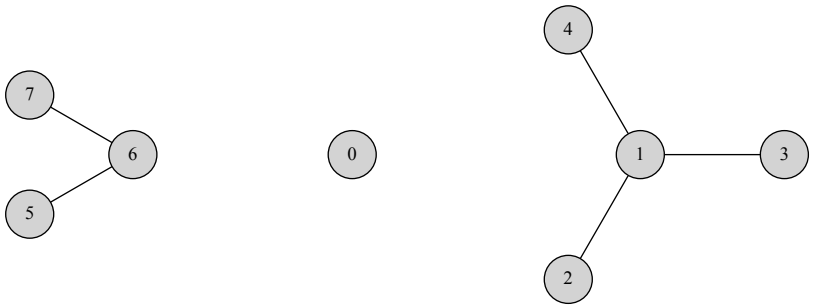


Figura 6.1. Bosque compuesto por tres árboles.

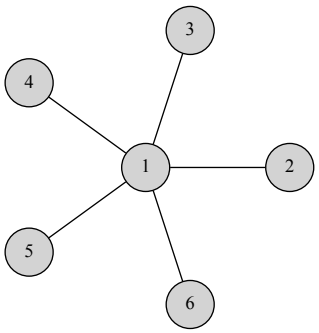


Figura 6.2. Árbol generador.

menores grafos conexos o los mayores grafos sin ciclos. En la Figura 6.1 y la Figura 6.2 tenemos ejemplos de estos conceptos.

A continuación, vamos a enunciar una serie de resultados acerca de las propiedades de estos elementos.

Lema 2. *Sea G un grafo conexo conteniendo un ciclo. Si eliminamos una de las aristas que forman el ciclo, entonces G sigue siendo conexo.*

Consecuencia de este lema, inmediatamente se puede demostrar el siguiente teorema.

Teorema 11. *Sea G un grafo conexo. Entonces G tiene un árbol generador.*

La demostración es muy sencilla.

Demostración. i. Si G es un grafo simple sin ciclos, entonces G es su propio árbol generador.

ii. Si G no es un grafo simple, gracias al Lema 2, podemos ir eliminando aristas de G hasta acabar con todos sus ciclos. De esta forma obtendríamos un subgrafo de G conexo y sin ciclos, es decir, un árbol generador. □

Proposición 17. *Sea G un árbol. Entonces G es un grafo plano.*

Demostración. Demostraremos que todo grafo sin ciclos de tamaño m es un grafo plano, por inducción sobre el número de aristas, m.

Si $m = 0$, el árbol no tiene aristas, por tanto, éstas no pueden cruzarse.

Supongamos el resultado cierto para árboles de tamaño menor que m.

Sea entonces G un grafo con m aristas y sin ciclos. Suprimamos una arista de G, entonces nos queda un grafo de tamaño $m-1$ sin ciclos, que sabemos que tiene una representación plana porque hemos supuesto el resultado cierto para todo árbol de tamaño menor que m. Como no tiene ciclos, no divide el plano en regiones, entonces puedo unir dos vértices cualesquiera por una arista sin cortar a ninguna otra. Por tanto, podemos añadir la arista eliminada teniendo una representación plana del grafo original. □

Nota 4. Para los siguientes resultados, recordemos que el orden y el tamaño de un grafo son el número de vértices y el número de aristas, respectivamente, que contiene.

Proposición 18. *Sea G un árbol finito de orden n con $n \geq 2$. Entonces G tiene un vértice de grado 1.*

Nota 5. Un vértice de grado 1, en los árboles, se denomina *hoja*.

Demostración. Como G tiene, al menos, dos nodos v_0, v_1, que podemos suponer adyacentes, escojo uno de ellos, sea éste, por ejemplo v_1. Al ser G un árbol, es conexo, por tanto, todos su vértices tienen, al menos, grado 1. Si $\mathrm{gr}(v_1) = 1$, hemos terminado. En caso contrario, $\mathrm{gr}(v_1) \geq 2$, luego existe un nuevo vértice v_2 unido a v_1. Estudiemos ahora v_2. Si es una hoja, hemos terminado. En caso contrario, es claro que v_2 no puede ser adyacente a v_0 o a v_1 con dos aristas paralelas, pues tendríamos ciclos en G y éste no sería un árbol, contradiciendo la hipótesis. Así, podemos concluir que existe un nuevo vértice v_3 conectado a v_2 distinto de v_0 y v_1. Siguiendo este procedimiento obtendremos una sucesión de vértices distintos pero, al ser G finito, necesariamente habrá un nodo de grado 1. □

Proposición 19. *Sea G un grafo de orden n. Entonces G es un árbol si, y sólo si, es conexo y tiene tamaño $n-1$.*

Demostración. Suficiencia: Sea G un árbol de orden $n = 1$. Entonces no puede contener ninguna arista, por tanto, $m = 0 = n-1$, verificando el resultado.

Supongamos ahora que G es un árbol de orden y tamaño m. Si $n \geq 2$, sabemos que G tiene al menos un vértice de grado 1. Eliminamos dicho vértice y, por tanto, la arista que incidía en él. Obtenemos así un grafo de $n-1$ vértices y $m-1$ aristas que sigue siendo un árbol. Aplicando este procedimiento inductivamente, obtendremos un árbol con un solo vértice, habremos eliminado $n-1$ vértices de G y, por tanto, $n-1$ aristas. Así, quedarán $m-(n-1)$ lados pero, al ser un árbol

de orden $n = 1$, sabemos que no puede contener aristas, por tanto, $m-(n-1) = 0$. De donde obtenemos que $m = n-1$, como queríamos.

Necesidad: Sea un grafo $G = (V, E)$ de orden n. Por hipótesis, sabemos que G tiene tamaño $n-1$ y es conexo, por lo que bastará demostrar que no contiene ciclos.

Si G tiene un solo vértice, entonces no tiene aristas y, por tanto, no contiene ciclos.

Si $n \geq 2$, como tiene $n-1$ aristas, entonces tiene al menos un vértice de grado 1 (de no ser así, es decir, si $\text{gr}(v) \geq 2$ para todo $v \in V$, tendríamos que $2(n-1) = \sum_{i=1}^{n} \text{gr}(v) \geq \sum_{i=1}^{n} 2 = 2n$, llegando a una contradicción). Eliminando dicho vértice y la arista que incidía en él, obtenemos un nuevo grafo de orden $n-1$ y tamaño $n-2$ conexo y conteniendo el mismo número de ciclos que G, ya que la arista eliminada no pertenecía a ningún ciclo. Si repetimos este procedimiento hasta llegar a un grafo conexo con un solo vértice y, por tanto, 0 aristas, éste último grafo tendrá los mismos ciclos que sus anteriores, en particular, los mismos ciclos que G. Como este último grafo no contiene ciclos, podemos afirmar que G tampoco los tendrá. Por tanto G es conexo y sin ciclos, G es un árbol. □

Teorema 12. *Sea $G = (V, E)$ un grafo simple de orden n. Son equivalentes:*

i. G es un árbol.

ii. Sean $u, v \in V$. Entonces existe un único camino simple conectando u y v.

iii. G es conexo, pero si suprimimos una de sus aristas deja de serlo.

iv. G no tiene ciclos, pero si le añadimos una arista, los tendrá.

v. G tiene tamaño $n-1$.

A continuación, una vez sentadas las bases de la teoría, podemos proceder finalmente a desarrollar algunos de los algoritmos más eficaces para la búsqueda de árboles y árboles generadores sobre un grafo dado.

Comenzaremos explicando algunos algoritmos para árboles etiquetados, es decir, árboles extraídos sobre grafos que tienen asignada una etiqueta para cada vértice.

A principios del siglo XIX, el químico francés J. L. Gay-Lussac descubrió que existían sustancias diferentes con la misma composición química, es decir, mismos átomos y en la misma cantidad, por lo que se pensó que la diferencia entre las sustancias residía en la forma que tenían de agruparse sus átomos. A mediados de siglo, dado el interés del problema, el matemático británico Arthur Cayley decidió aplicar sus conocimientos en la Teoría de Grafos sobre este campo, estudiando los isómeros de alcanos. Para ello, identificó cada alcano con un árbol, tomando como nodos los átomos de carbono y como lados los enlaces entre los mismos. De esta forma, años de estudio después, Cayley consiguió obtener una fórmula para contabilizar el número de isómeros alcanos para un número dado de átomos de carbono, lo que se tradujo en una fórmula sorprendentemente simple para contar cuántos árboles etiquetados distintos hay de orden n.

Teorema 13 (Fórmula de Cayley.)**.** *Sea $n \in \mathbb{N}, n \geq 2$. Entonces el número de árboles etiquetados distintos de n vértices es n^{n-2}.*

Demostración. Demostraremos la Fórmula de Cayley a través del Código de Prüfer.

Sea $G = (V, E)$ un árbol etiquetado con $V = \{1, 2, \ldots, n\}$ y sean $u_1 \in V$ la menor hoja de G y v_1 el nodo adyacente a u_1 en G. Tomaremos $G = G_1$.

Sean ahora u_2 la menor hoja del árbol $G_2 = G_1 \setminus \{u_1\}$ y v_2 el nodo adyacente a u_2 en G_2. Es decir, partiendo de G_1, para todo $k = 1, \ldots, n$, tomaremos u_k como la menor hoja en G_k y v_k como su vértice adyacente en G_k. Para cada $k = 2, \ldots, n-1$, haremos $G_{k+1} = G_k \setminus \{u_k\}$.

De esta forma, podemos listar las $n-1$ aristas de G como sigue:

$$\begin{pmatrix} u_1 & u_2 & u_3 \cdots & u_{n-1} \\ v_1 & v_2 & v_3 \cdots & v_{n-1} \end{pmatrix}. \tag{6.1}$$

Así, el código de Prüfer de G es $C = [v_1, v_2, v_3, \cdots, v_{n-2}]$. Es claro que la asignación $G \to [v_2, v_3, v_4, \cdots, v_{n-2}]$ está bien definida, por lo que queda demostrar que cada $n-2$-upla de elementos en $\{1, 2, \ldots, n\}$ es el Código Prüfer de un árbol generador de $\mathbb{K}_n$.

Necesariamente deber ser $v_{n-1} = n$. Entonces:

- $V_{G_k} = \{u_k, u_{k+1}, \ldots, u_{n-1}, n = v_{n-1}\}$, para todo $k = 1, 2, \ldots, n$.
- $E_{G_k} = \{(u_i, v_i), k \leq i \leq n-1\}$.

Es claro que $\mathrm{gr}(v), v \in V$ es el número de apariciones de v en la lista de aristas de G. Al aparecer v exactamente una vez en $(u_1, u_2, u_3, \cdots, u_{n-1}, v_{n-1})$, tenemos entonces que v aparece exactamente $\mathrm{gr}(v) - 1$ veces en $(v_1, v_2, v_3, \cdots, v_{n-2})$.

En particular, las hojas de G son los elementos de $V = \{1, 2, \ldots, n\}$ que no se encuentran en $\{v_1, v_2, v_3, \cdots, v_{n-2}\}$, por tanto, u_1 está unívocamente determinado por ser el menor elemento de $V = \{1, 2, \ldots, n\}$ no contenido en $(v_1, v_2, v_3, \cdots, v_{n-2})$.

De la misma forma, si $v \in V_{G_k}$, entonces aparece exactamente $\mathrm{gr}_{G_k}(v) - 1$ veces en $(v_k, v_{k+1}, \ldots, v_{n-1})$. Por tanto, las hojas de G_k son los elementos de $V = \{1, 2, \ldots, n\}$ que no aparecen en $\{u_1, u_2, \ldots, u_{k-1}, v_k, \ldots, v_{n-2}\}$. Existe al menos un nodo satisfaciendo estas condiciones, el menor de ellos será u_k.

En virtud de lo anterior, cada $n-2$-upla de elementos de $V = \{1, 2, \ldots, n\}$ es código Prüfer de un único árbol de orden n. □

Así pues, el primer algoritmo que estudiaremos en esta sección es el *Código de Prüfer*, herramienta que ha servido para demostrar la fórmula de Cayley [13].

A.6 Código Prüfer de un árbol

Sea T un árbol etiquetado de orden n, se denomina código Prüfer de dicho árbol al código numérico de longitud $n-2$ obtenido como sigue:

1. Se parte de la lista vacía $P = [\,]$ y del árbol etiquetado T.

2. Si T tiene sólamente dos vértices, se devuelve P y se termina el proceso.

3. Sea u el vértice de grado 1 de menor etiqueta de T y sea v el vértice adyacente a u. Se añade v a P y se eliminan el vértice u y la arista (u,v) de T.

4. Vuelta al paso dos con el nuevo código P y el árbol resultante T.

Veamos a continuación un ejemplo para entender cómo funciona este algoritmo.

Supongamos que queremos obtener el Código Prüfer para el grafo G de la Figura 6.3.

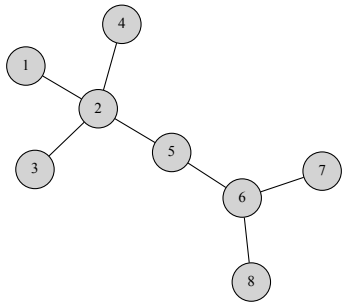
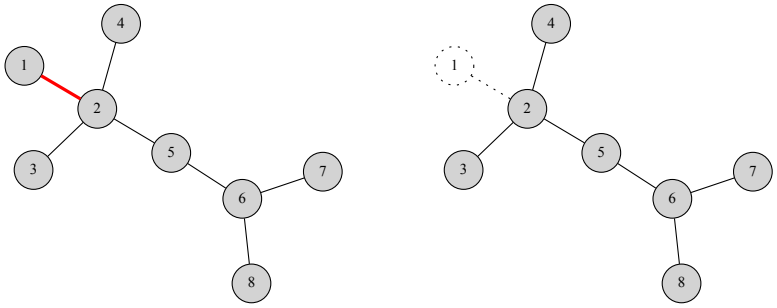


Figura 6.3

En primer lugar, debemos cotejar que, efectivamente, dicho grafo G es un árbol. Vemos que G es un grafo conexo que no contiene ciclos, podemos entonces afirmar que G es un árbol.

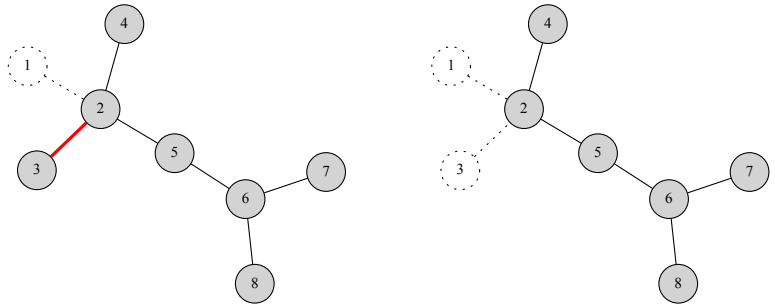
Procedemos entonces a implementar el algoritmo. Inicializamos una lista $P = []$ que será nuestro Código Prüfer.

Ahora buscamos, de entre los vértices de grado 1, el de menor etiqueta, en este caso, el nodo 1. Marcamos entonces la arista $(1,2)$ y añadimos a nuestra lista P, el nodo 2. Borramos finalmente dicho vértice y, por tanto, la arista que incidía sobre él.



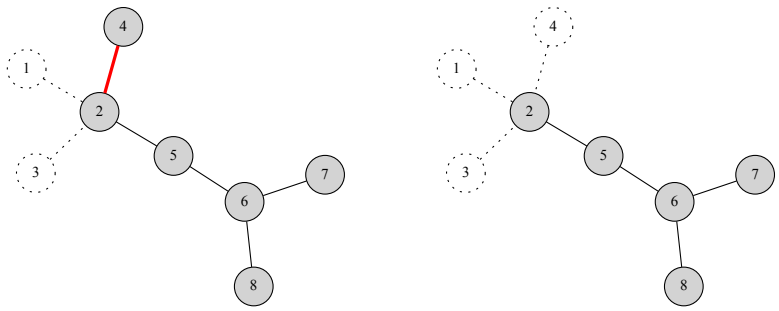
$$P = [2]$$

Repetimos ahora el proceso con el grafo resultante y la lista $P = [2]$. En este caso, el vértice de grado 1 de menor etiqueta es el nodo 3, con lo cual, señalamos la arista $(3,2)$ y añadimos a P el nodo 2. Borramos a continuación el vértice 3 y, por tanto, la arista $(3,2)$.

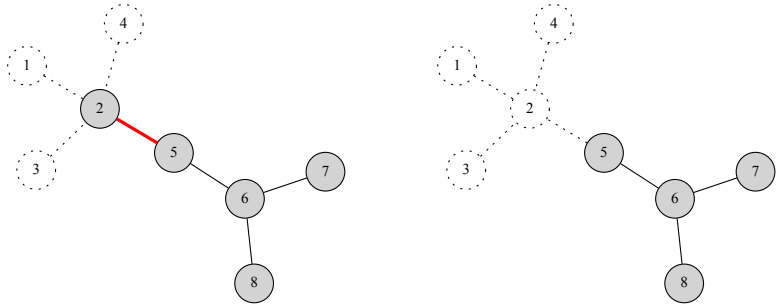


$$P = [2,2]$$

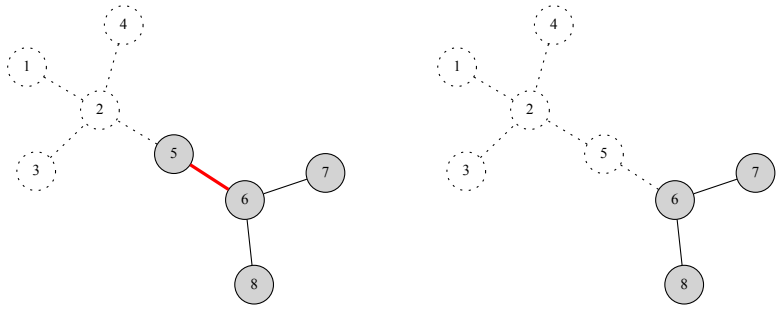
Así pues, reiteramos este procedimiento hasta obtener un grafo resultante con una sola arista:



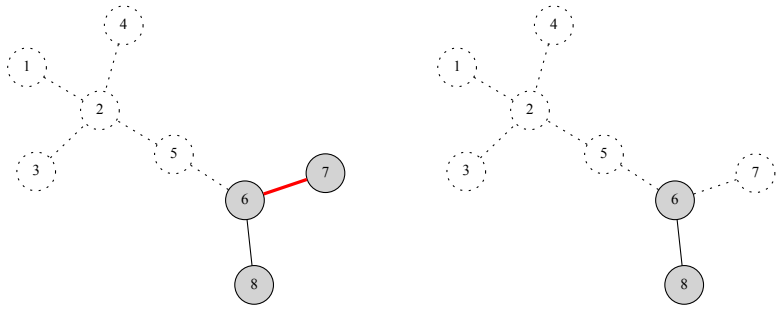
$$P = [2,2,2]$$



$P = [2,2,2,5]$



$P = [2,2,2,5,6]$



$P = [2,2,2,5,6,6]$

Llegados a este punto, como nuestro grafo resultante tiene sólo dos vértices, devolvemos la lista $P = [2,2,2,5,6,6]$ como el código Prüfer asociado a dicho árbol.

En nuestro repositorio [7], en la clase *grafos*, podemos encontrar la implementación de este algoritmo como un método. Lo hemos aplicado, a modo de ejemplo, sobre el grafo de la Figura 6.4.

El código para su ejecución es el siguiente:

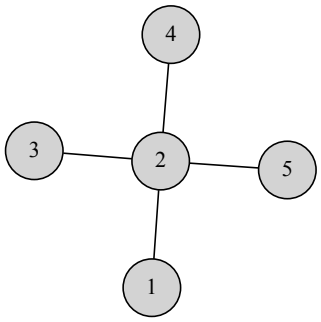


Figura 6.4

```
from Prufer import *

g=Grafo([],[(1,2),(2,3),(2,4),(2,5)])

g.Prufer()
```

La salida que obtenemos por pantalla es la lista `[2,2,2]`, el Código de Prüfer asociado a ese grafo. Es importante mencionar que, si el grafo estudiado no es un árbol, el método Prüfer retornará un error por pantalla.

De la misma forma, debemos saber revertir el proceso, es decir, construir un árbol etiquetado a partir de su Código Prüfer, pues tan importante es saber cuántos árboles etiquetados hay como poder obtenerlos. Así, procedamos ahora a enunciar los pasos de este algoritmo.

A.7 De Código Prüfer a árbol

1. Como elementos de partida tenemos el conjunto de etiquetas $V = \{1, \ldots, n\}$, el Código Prüfer P, de tamaño $n-2$, y el grafo vacío T.

2. Si P es vacío, entonces V tiene dos elementos. Se añade a T la arista que une dichos elementos y se termina el proceso.

3. Se elige el menor número $v \in V$ tal que $v \notin P$, se añade una arista a T entre v y el primer elemento de P. Se elimina v de V y se elimina el primer elemento de P.

4. Se vuelve a 2 con el nuevo código P, el nuevo grafo T y el nuevo conjunto de etiquetas V.

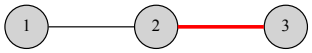
Veamos la reconstrucción del código P obtenido en el algoritmo anterior para esclarecer el funcionamiento de este algoritmo.

La longitud de P es 6, por tanto, el algoritmo nos dice que nuestro grafo T tendrá 8 vértices. El punto de partida será entonces un grafo vacío T, la lista $P = [2,2,2,5,6,6]$ y el conjunto de vértices $V = \{1,2,3,4,5,6,7,8\}$.

Así, el primer paso es escoger el vértice de menor etiqueta de V que no está en P y unirlo con el primer elemento de P. Es decir, en este primer paso, añadimos la arista $(1,2)$ y borramos el primer elemento de P, 2, y el elemento escogido de V, 1.

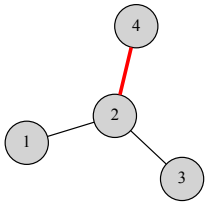
$$P = [2,2,5,6,6] \quad V = \{2,3,4,5,6,7,8\}$$

Repetimos el proceso con los T, P y V resultantes. En este caso, añadiríamos la arista $(2,3)$ y borraríamos el primer elemento de P, 2, y el elemento escogido de V, 3.

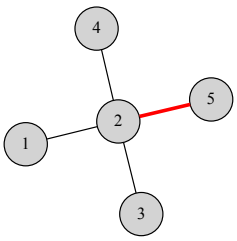


$$P = [2,5,6,6] \quad V = \{2,4,5,6,7,8\}$$

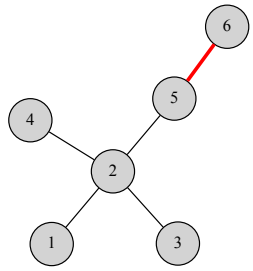
Repetimos este proceso hasta que P se quede sin elementos.



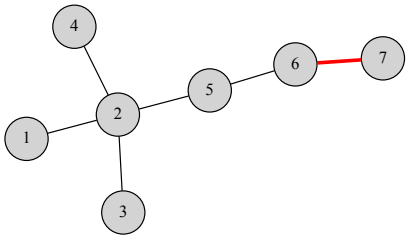
$$P = [5,6,6] \quad V = \{2,5,6,7,8\}$$



$P = [6,6] \quad V = \{5,6,7,8\}$

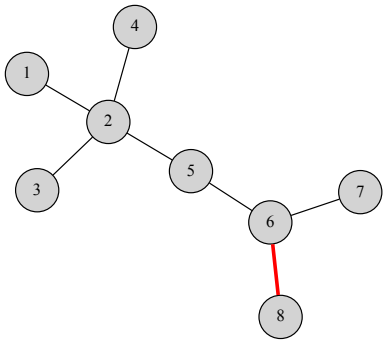


$P = [6] \quad V = \{6,7,8\}$



$P = [\,] \quad V = \{6,8\}$

Llegados a este punto, donde P ya no tiene más elementos, añadimos a T la última arista, la cuál conectará los dos últimos elementos de V.



Como podemos comprobar, el grafo obtenido T es un árbol.

Podemos encontrar una implementación de este algoritmo en nuestro repositorio [7]. A modo de ejemplo, lo hemos ejecutado sobre el código Prüfer del grafo de la Figura 6.4.

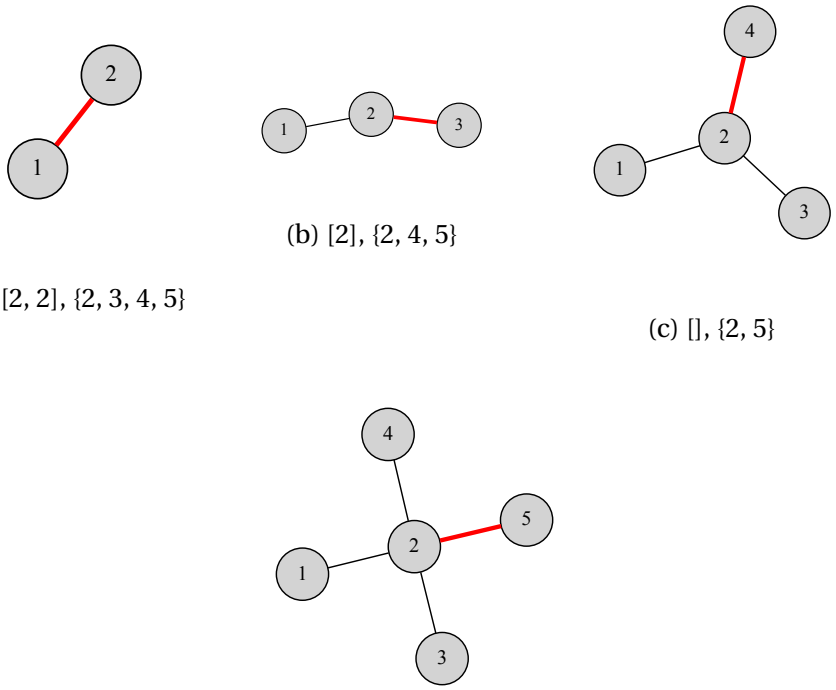
El código a ejecutar es:

```
from Prufer import *

g=Grafo([],[(1,2),(2,3),(2,4),(2,5)])

P = g.Prufer()
Prufer_a_Arbol(P, True)
```

Y la salida por pantalla la encontramos en la Figura 6.5.



(a) [2, 2], {2, 3, 4, 5}

(b) [2], {2, 4, 5}

(c) [], {2, 5}

Figura 6.5. De código Prüfer a árbol.

A continuación, vamos a centrarnos en la búsqueda de árboles generadores. Expondremos los algoritmos de dos estrategias "inversas" para, dado un grafo conexo de orden n y tamaño m, obtener un árbol generador.

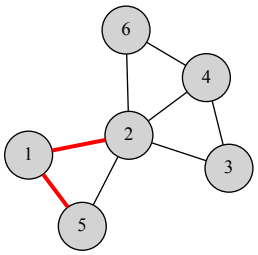
La ventaja de extraer árboles generadores de un grafo reside en el hecho de que permite conectar todos los nodos del grafo con el mínimo número de aristas. Es decir, los árboles representarían situaciones en las que no importa el coste de recorrer sus lados, sino solamente el de construcción.

A.8 Algoritmo Building-up (Constructivo)

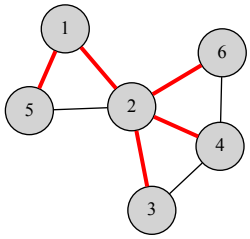
Se van escogiendo aristas del grafo original, de una en una, y de forma que no vayan formando ciclos con las ya escogidas, hasta tener un grafo de tamaño $n-1$, siendo n el orden del grafo original. Dicho grafo será el árbol generador.

De nuevo, ejemplifiquemos el funcionamiento de este algoritmo, buscando el árbol generador del grafo de la Figura 4.3.

Vamos escogiendo aristas de una en una, por ejemplo, la $(1,5)$ y la $(1,2)$.



Llegados a este punto, deberíamos escoger cualquier arista exceptuando la $(5,2)$, que formaría un ciclo con las que ya tengo. Así, elegiríamos por ejemplo, las aristas $(2,3),(2,4)$ y $(2,6)$.



Por un lado, vemos que cualquier arista que añadiésemos formaría un ciclo, con lo que damos por terminado el algoritmo pero, además, vemos que el número de aristas de nuestro nuevo grafo es menor en una unidad al número de vértices, por tanto, tenemos una condición más para saber que hemos terminado el proceso. Es claro que el grafo obtenido es un árbol generador.

La implementación de este algoritmo la podemos encontrar en nuestro repositorio [7]. Hemos tomado como ejemplo, el grafo de la Figura 6.6.

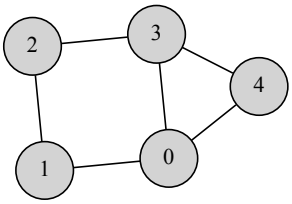


Figura 6.6

El código para su ejecución es:

```
from Constructivo_Destructivo import *

g=Grafo([],[(0,1),(1,2),(2,3),(3,4),(0,3),(0,4)])

Constructivo(g, True)
```

Y la salida por pantalla se corresponde con la Figura 6.7.

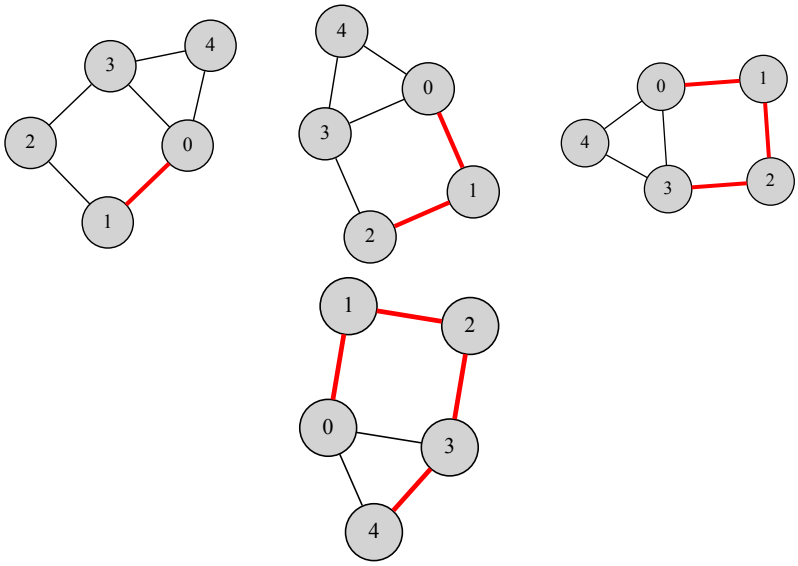


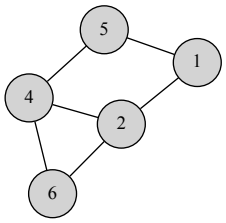
Figura 6.7. Algoritmo Building Up.

A.9 Algoritmo Cutting-down (Destructivo)

Estrategia muy similar a la anterior, pero en este caso, vamos eliminando aristas del grafo original, de una en una y de forma que cada arista eliminada rompa un ciclo. Se descartarán entonces $m-(n-1)$ aristas, siendo m y n el tamaño y el orden del grafo estudiado, respectivamente. El grafo resultante será el árbol generador.

Veamos cómo procederíamos en el grafo de la Figura 6.8 con este algoritmo.

Observamos que tenemos varios ciclos en nuestro grafo de partida, escogemos cualquiera de ellos, por ejemplo, el ciclo $[1,5,2,1]$. Ahora, elegimos cualquier arista de ese ciclo y la eliminamos para romperlo. Sea la arista elegida, por ejemplo, la $(2,5)$.



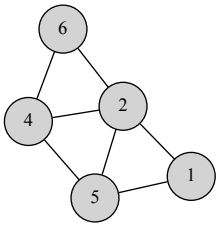
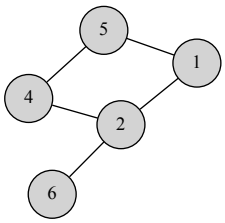
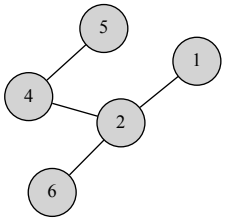


Figura 6.8

Buscamos ahora otro ciclo del grafo resultante, sea éste, por ejemplo, el ciclo $[2,4,6,4]$ y eliminemos de él la arista $(4,6)$.



Seguimos procediendo del mismo modo, ahora solo tenemos el ciclo $[1,2,4,5,1]$. Eliminemos de él la arista $(1,5)$, por ejemplo.



Hemos obtenido así un grafo sin ciclos, por lo que daríamos por terminado el proceso. En efecto, el algoritmo nos dice que el proceso termina al eliminar $8-(6-1)=3$ aristas, como hemos hecho. Además, es claro que el grafo obtenido es un árbol generador.

De nuevo, al igual que en el algoritmo anterior, podemos visualizar la implementación de este algoritmo aplicado, a modo de ejemplo, sobre el grafo de la Figura 6.6 en nuestro repositorio [7].

El código para su ejecución es:

```
from Constructivo_Destructivo import *

g=Grafo([],[(0,1),(1,2),(2,3),(3,4),(0,3),(0,4)])

Destructivo(g, True)
```

Y la salida que obtendremos por pantalla la encontramos en la Figura 6.9.

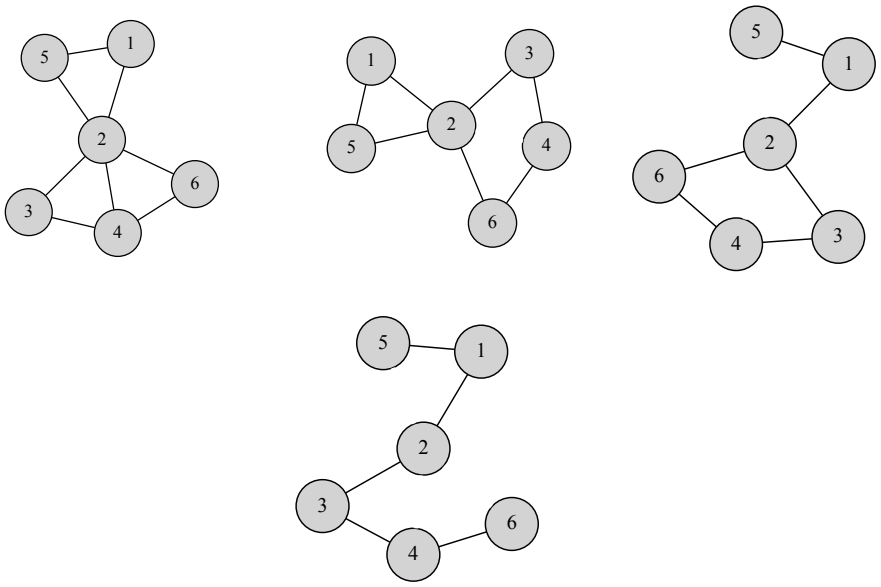


Figura 6.9. Algoritmo Cutting Down.

A continuación, vamos a centrarnos en algoritmos de búsqueda de elementos de gran importancia hoy día, los árboles generadores de peso mínimo. Para ello, debemos incluir el siguiente concepto.

Definición 28. Sea G un grafo ponderado, se dice que T es un *árbol generador de peso mínimo* si cumple las siguientes condiciones:

i. T es un árbol generador de G.

ii. De entre todos los árboles generadores de G, la suma total de los pesos de las aristas de T es la menor.

Teorema 14. *Sea G un grafo finito ponderado, entonces G contiene, al menos, un árbol generador de peso mínimo.*

Demostración. Suponemos G un grafo finito, entonces contiene un número finito de subgrafos y, por tanto, un número finito de árboles generadores. De esta forma, el conjunto S, de las sumas totales de los valores de las aristas de cada árbol generador, tiene mínimo. □

Es decir, los árboles de peso mínimo, además de mantener conectada toda una red de puntos, como ya hacían los árboles generadores, lo hacen de la forma más económica, considerando el peso de las aristas como el coste al construirla, al recorrerla, etc., este tipo de árboles nos proporcionan la opción más barata.

Esta clase de grafos son muy utilizados en la construcción de redes eléctricas, telefónicas, de ordenadores, etc. Imaginemos que tenemos un árbol generador de peso mínimo x, entonces podemos asegurar que nos desplazaremos entre dos vértices cualesquiera del árbol con un coste máximo de x.

A.10 Algoritmo de Kruskal constructivo

Se aplica el algoritmo *building-up* antes expuesto, pero escogiendo cada vez la arista de menor peso posible.

Veamos un ejemplo del funcionamiento de este algoritmo, apliquémoslo sobre el grafo de la Figura 6.10.

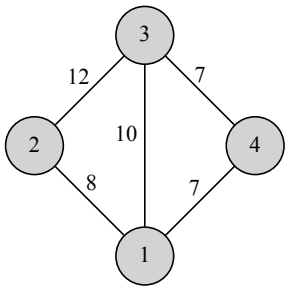
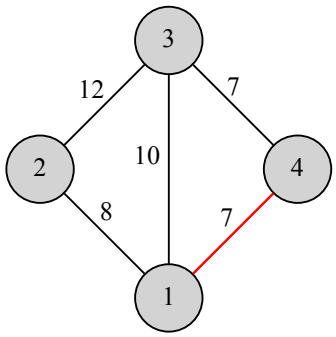
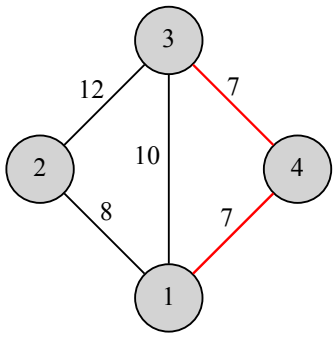


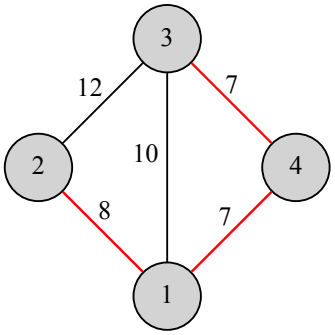
Figura 6.10

Ya sabemos cómo trabaja el algoritmo *Building-up*, con lo cual, solo debemos ir seleccionando las aristas de menor peso. Para el primer paso, vemos que las aristas $(1,4)$ y $(3,4)$ tienen peso 7. En esta situación, elegimos cualquiera de ellas. De hecho, en el siguiente paso, añadiremos la arista restante de peso 7, pues no forma ciclo con la anteriormente escogida.





A continuación cogeríamos la arista $(1,2)$, de peso 8.



Y éste es el grafo final, puesto que, cualquiera de las dos aristas restantes, formaría un ciclo con las ya incluidas. Además, como dice el algoritmo, hemos añadido $4-1=3$, aristas, por lo que hemos terminado. De hecho, sabemos por la *Proposición* 20, que el grafo final es efectivamente un árbol generador y es de peso mínimo.

En nuestro repositorio [7] ofrecemos la implementación de este algoritmo. A modo de ejemplo lo hemos aplicado sobre el grafo de la Figura 6.11.

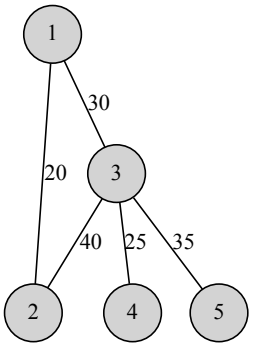


Figura 6.11

El código a ejecutar es el siguiente:

```
from Kruskal_Constructivo_Destructivo import *

g=Grafo([],[(1,2),(1,3),(2,3),(3,4),(3,5)])

g.ponderado([20,30,40, 25, 35])

Kruskal_Constructivo(g, True)
```

Obteniéndose por pantalla la salida que encontramos en la Figura 6.12.

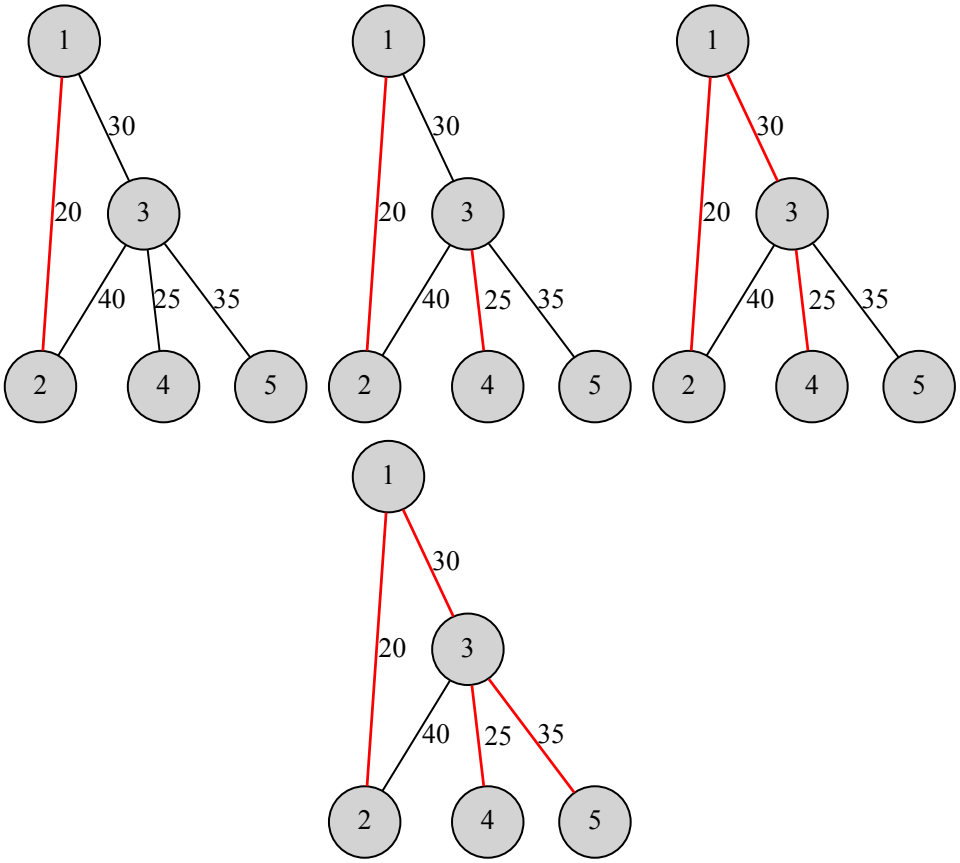


Figura 6.12. Algoritmo de Kruskal constructivo.

A.11 Algoritmo de Kruskal destructivo

De nuevo, aplicamos la estrategia *cutting-down* eliminando cada vez la arista de mayor peso posible.

Apliquemos este algoritmo al grafo de la Figura 6.10. Deberíamos de obtener el mismo árbol generador que con el algoritmo de Kruskal constructivo.

Este algoritmo nos dice que debemos eliminar las $5-(4-1)=2$ aristas de mayor peso del grafo original, las cuales son la $(2,3)$ y la $(3,1)$, en ese orden.

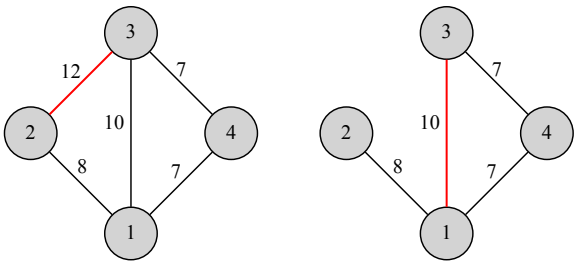
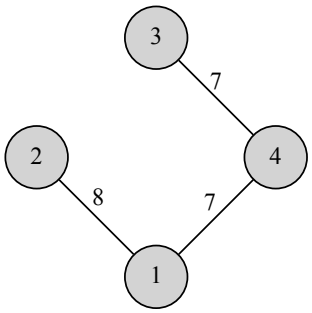


Figura 6.13

Así, el grafo obtenido es:



Como queríamos. Es claro que dicho grafo es un árbol y es de peso mínimo.

De nuevo, en nuestro repositorio [7], observamos el funcionamiento de este algoritmo, el cuál hemos ejecutado, a modo de ejemplo, sobre el grafo de la Figura 6.11.

El código para su ejecución es:

```
from Kruskal_Constructivo_Destructivo import *

g=Grafo([],[(1,2),(1,3),(2,3),(3,4),(3,5)])

g.ponderado([20,30,40, 25, 35])

Kruskal_Destructivo(g, True)
```

Y la salida que obtenemos por pantalla la encontramos en la Figura 6.14

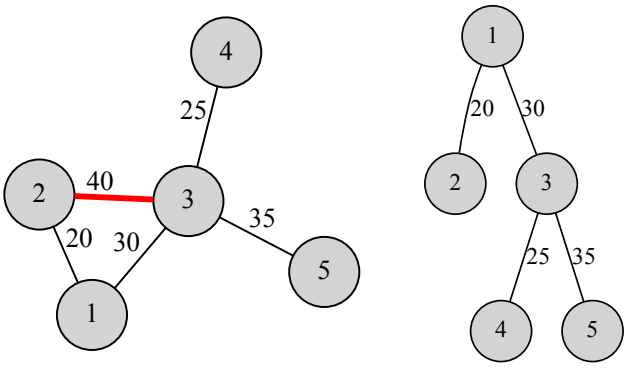


Figura 6.14. Algoritmo de Kruskal destructivo.

A.12 Algoritmo de Prim

1. Se crean los conjuntos $V = []$ y $E = []$ donde incluiremos los vértices y aristas que determinan el árbol, respectivamente.

2. Se escoge la arista de menor peso y se marca uno de sus vértices, llamémoslo v, el cuál incluimos en V. Se tiene entonces $V = [v]$.

3. En cada iteración, se incluye en el conjunto V un vértice w y en el conjunto E una arista e que satisfagan lo siguiente:

   - Que el vértice w no se encuentre ya en el conjunto de vértices V.
   - Que w sea adyacente, mediante e, a un vértice del conjunto V.
   - Que e no forme ciclos con el resto de aristas del conjunto E.
   - Que e sea la arista de menor peso de entre las no escogidas aún.

4. Si el número de pasos son $n-1$, se termina el proceso.

Ejemplifiquemos este algoritmo aplicándolo sobre el grafo de la Figura 6.15.

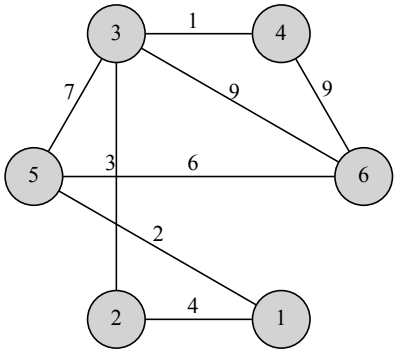
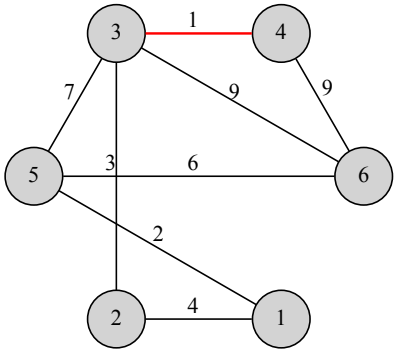
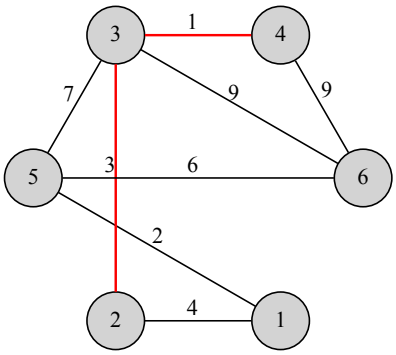


Figura 6.15

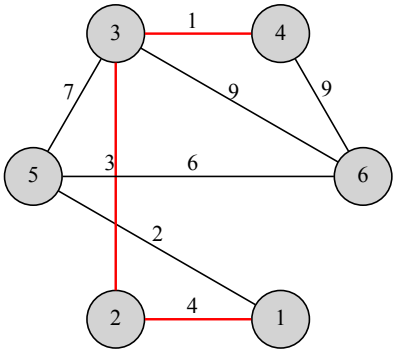
Comenzamos eligiendo la arista de menor peso, en este caso, elegimos la $(3,4)$, de peso 1. En caso de haber dos aristas con el mismo peso, peso mínimo, se escoge cualquiera de ellas.

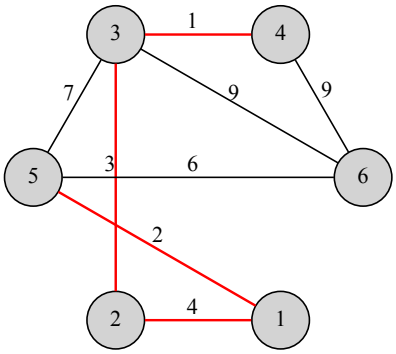


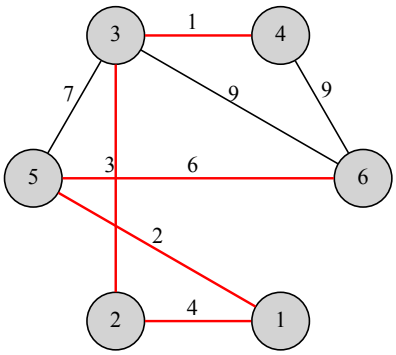
A continuación, encontramos que la siguiente arista candidata a ser elegida es la $(1,5)$, por tener peso 2. Sin embargo, el algoritmo dice que la arista escogida debe ser incidente en uno de los vértices ya escogidos, por tanto, debemos seleccionar la arista $(2,3)$, de peso 3.



Así pues, seguimos realizando este procedimiento hasta haber seleccionado $6-1=5$ aristas.







Concluimos, por la *Proposición* 20 que el grafo obtenido es un árbol generador y es de peso mínimo.

Podemos encontrar en nuestro repositorio [7] la implementación de este algoritmo. Lo hemos ejecutado sobre el grafo de la Figura 6.16, para ejemplificarlo.

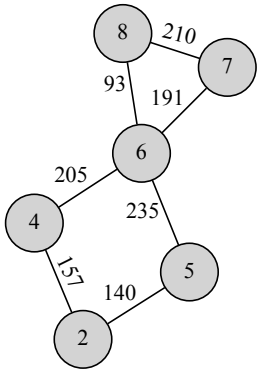


Figura 6.16

El código a ejecutar es el siguiente:

```
from Prim import *

g=Grafo([],[(1,2),(2,3),(1,4),(3,4),(4,5),(4,6),(5,6)])

g.ponderado([157,140,205,235,191,93,210])

Prim(g, True)
```

Y la salida por pantalla se corresponde con la imagen de la Figura 6.17.

La validez de los algoritmos de KrusKal y Prim [8] queda probada con el siguiente resultado, para el cual necesitamos primero la siguiente definición.

Definición 29. Sea $G = (V, E)$ un grafo y sea el conjunto $V' \subset V$. Se denomina *cociclo asociado al conjunto* V', y lo denotaremos por $C(V')$, al conjunto formado por las aristas que inciden en un vértice de V' y en otro de $V \setminus V'$.

Proposición 20. *Sea* $G = (V, E)$ *un grafo ponderado con función de pesos* $w : E \to \mathbb{R}$ *y* $T = (V, E')$ *un árbol generador de* G. *Son equivalentes:*

i. T *es un árbol generador de peso mínimo.*

ii. *Sea la arista* $e \in E \setminus E'$ *y sea* $e_1 \in E$ *una arista cualquiera integrante del único ciclo existente en el grafo* $(V, E' \cup \{e\})$. *Entonces se cumple que* $w(e_1) \leq w(e)$ *donde* $w(e)$ *es el peso asociado a la arista* e, *para todo* $e \in E$.

iii. *Para todo* $e_1 \in E'$, *se tiene que* $w(e_1) \leq w(e)$, *para todo* $e \in C(V_1^{e_1})$, *donde* $C(V_1^{e_1})$ *es el cociclo asociado a cualquier componente conexa de* $(V, E' \setminus \{e_1\})$.

Demostración.

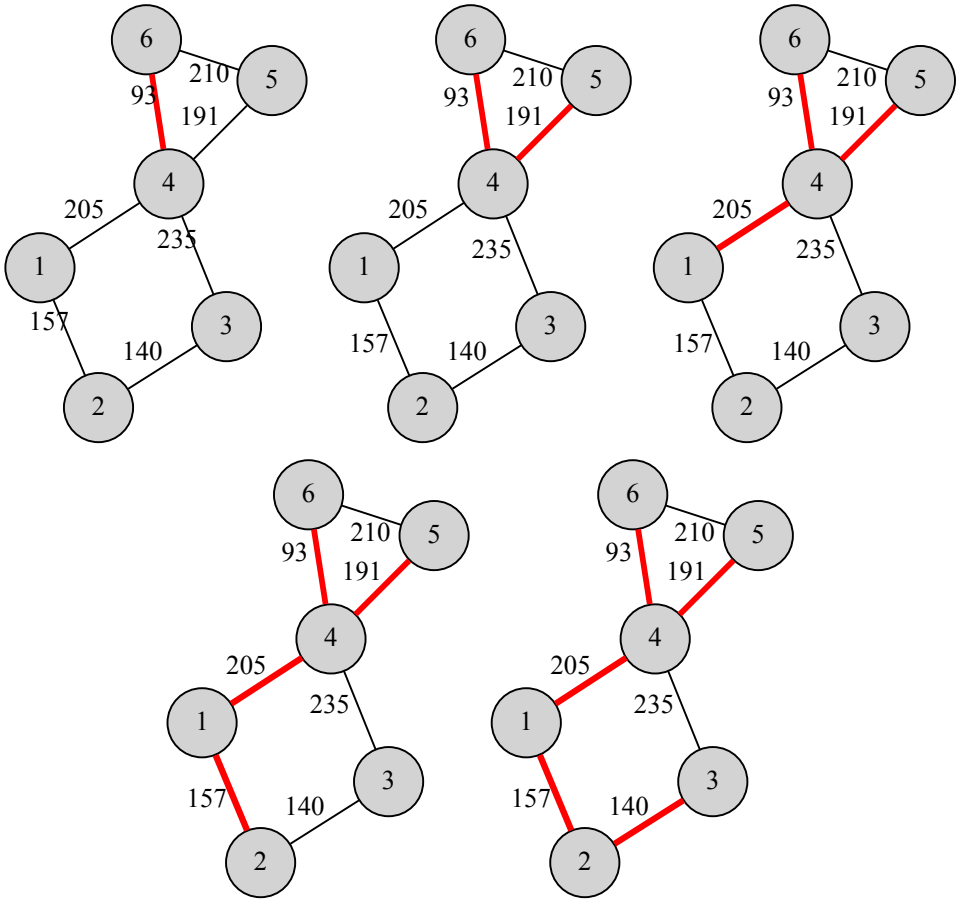


Figura 6.17. Algoritmo de Prim.

i. → ii. Supongamos el árbol generador de peso mínimo $T = (V, E')$ y supongamos que $\exists\ e \in E \setminus E', w(e) < w(e_1)$ donde e_1 pertenece al único ciclo del grafo $G' = (V, E' \cup \{e\})$. Entonces, eliminando la arista e_1 de G', obtendríamos un árbol generador con un peso menor a T, llegando así a una contradicción.

ii. → iii. Sea la arista $e_1 \in E'$. Eliminando e_1 descomponemos T en dos componentes conexas. Sea entonces $e \in C(V_1^{e_1})$. Si añadimos e a T, obtenemos un árbol generador de G, añadiendo entonces, de nuevo, la arista e_1, se forma un ciclo que contiene a la arista e, por tanto, por hipótesis, $w(e_1) \leq w(e)$.

iii. → i. Consideramos el árbol generador de peso mínimo en G, $T' = (V, E'')$ y supongamos el árbol $T = (V, E')$ satisfaciendo las condiciones de iii. Si $T = T'$, la demostración es trivial. En caso contrario, podemos afirmar que existe $e_1 \in E' \setminus E''$. Si eliminamos e_1 de T, obtenemos dos componentes conexas, sean éstas $V_1^{e_1}$ y $V \setminus V_1^{e_1}$. Si añadimos e_1 a T' se formará un ciclo en T', además de un camino c, con $e_1 \notin c$, que conecta un vértice de $V_1^{e_1}$ con uno de $V \setminus V_1^{e_1}$, por tanto, existirá una arista $e \in C(V_1^{e_1})$. Entonces, por hipótesis, se tendrá que $w(e) \geq w(e_1)$. Por otro lado, T' es un árbol generador de peso mínimo, por lo que $w(e) \leq w(e_1)$, pues, de no ser así, eliminando e y añadiendo e_1, obtendríamos un árbol generador de

menor peso. Por tanto, podemos concluir que $w(e) = w(e_1)$. Reemplazamos ahora en T la arista e por e_1 y obtenemos así un árbol generador de igual peso que T y una arista más en común con T'. Reiterando este procedimiento, obtendremos una sucesión de árboles generadores de igual coste a T, hasta llegar a T', por lo que $w(T) = w(T')$. □

A.13 Algoritmo de Boruvka

Este algoritmo puede formar ciclos si hay aristas con el mismo peso, en tal caso, debe establecerse un orden total entre las aristas del grafo y, en caso de tener que elegir entre dos aristas de igual peso, escoger la primera según ese orden. Veamos cómo funciona para un grafo de orden n.

1. Se recorren los vértices del grafo estudiado de uno en uno en cualquier orden y se elige la arista de menor peso que incida en él, con una condición: si un vértice ya ha sido marcado al ser el extremo de una arista escogida, ese vértice ya no se recorre.

2. Una vez terminado el paso 1, tenemos como resultado m componentes conexas. Se estudian entonces todas las aristas de menor peso que unan una componente conexa con otra y se escoge la de menor peso, en caso de haber dos con igual peso, como ya hemos dicho, se establece un orden y se escoge la primera por dicho orden.

3. El proceso termina cuando el grafo resultante es conexo.

Veamos cómo funciona el algoritmo de Boruvka aplicándolo sobre el grafo de la Figura 6.15.

Empezamos a recorrer los vértices del grafo, por ejemplo, por orden creciente de su etiqueta. Sigamos el procedimiento con la Figura 6.18. De las dos aristas incidentes en el nodo 1, vemos que la de menor peso es la arista $(1,5)$, por tanto, la añadimos a T y el nodo 5 ya no será recorrido por haber sido ya incluido. Seguimos con el vértice 2, cuya arista incidente de menor peso es $(2,3)$, la añadimos a T y ya no recorreremos el nodo 3. Nos posicionamos en el nodo 4, escogemos la arista $(4,3)$ y, finalmente, nos vamos al vértice 6 y seleccionamos la arista $(6,5)$.

Así pues, ya hemos completado el primer paso del algoritmo, quedando el grafo T como un grafo disconexo con dos componentes conexas, como observamos en el Figura 6.19.

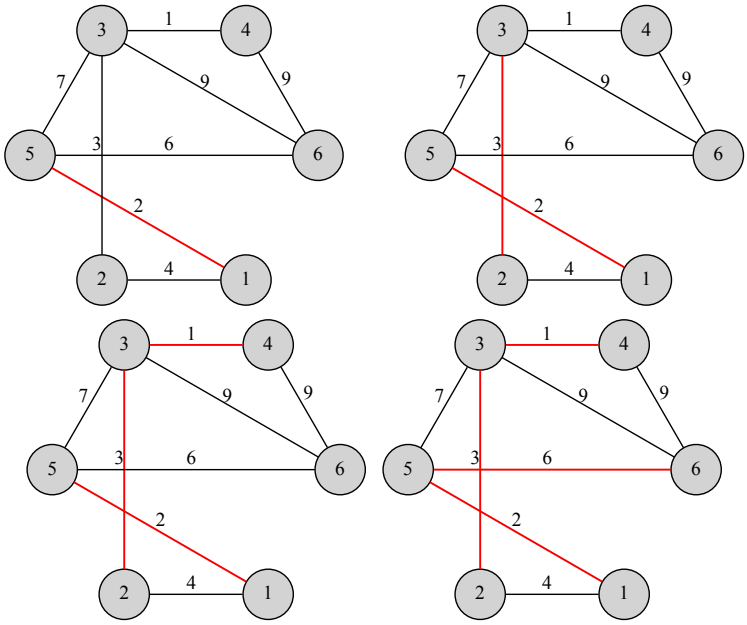


Figura 6.18

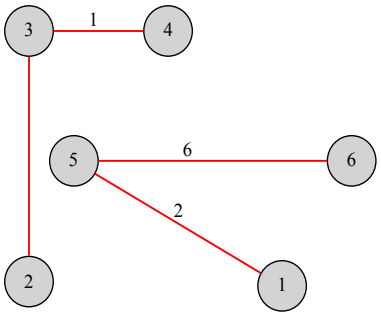
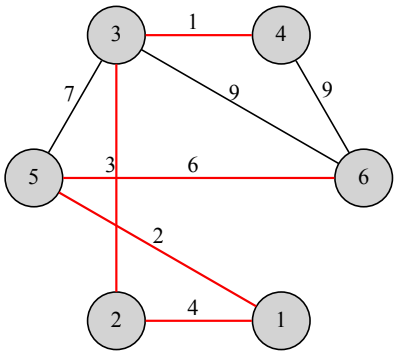


Figura 6.19

El siguiente paso consiste en cotejar las distintas aristas que me conectan ambas componentes conexas y elegir aquella de menor peso. En nuestro caso, la arista $(1,2)$ es la que tiene menor peso y hace de T un grafo conexo al añadirla.

Así, el árbol generador de peso mínimo del grafo inicial es:



La validez del algoritmo de Boruvka [9] queda probada con los siguientes resultados.

Lema 3. *Sean un grafo $G = (V, E)$ y un vértice $v \in V$. Entonces el árbol generador de peso mínimo T de G debe contener la arista (v, w) que es la arista de menor peso incidente en v.*

Demostración. Supongamos que dicha arista de peso mínimo (v, w) no esta contenida en T, por ser T un árbol, debe haber un arista incidente en v, sea ésta (v, u). Añadiendo ahora la arista (v, w) a T, de nuevo, por ser T, obtendríamos un ciclo que pasa por v. Como (v, w) es de menor peso que (v, u), deberíamos borrar esta última y mantener (v, w), obteniendo así un árbol generador T' de G de menor peso que T. Llegamos así a una contradicción ya que T es el árbol generador de peso mínimo. Concluimos pues que el lado (v, w) debe estar en T. □

Lema 4. *El conjunto de aristas escogidas en el paso 1 del algoritmo de Boruvka inducen un bosque en el gafo G sobre el que se aplica.*

Demostración. Por el *Lema* 3, cada una de las aristas escogidas en el paso 1 del algoritmo, debe ser una arista del árbol generador de peso mínimo de G. Durante cada iteración en este paso 1 del algoritmo, las aristas escogidas forman un subgrafo del árbol generador de peso mínimo de G. Por tanto, esas aristas pueden no estar conectadas entre sí, pero no forman ciclos, así, forman un bosque en G. □

Lema 5. *Sea G' el grafo obtenido al aplicar el paso 1 del algoritmo de Boruvka sobre un grafo G. Entonces el árbol generador de peso mínimo de G es la unión de las aristas escogidas durante dicho paso con las aristas del árbol generador de peso mínimo del grafo G'.*

Demostración. Por el *Lema* 4, sabemos que tras el paso 1 del algoritmo obtenemos un bosque de G y esas aristas pertenecen al árbol generador de peso mínimo de G. Cada arista escogida en el paso 1 marca un vértice distinto de G, por lo que, contrayendo cada componente conexa en un sólo vértice, los vértices que permanecen, también permanecen en G'. La unión del árbol generador de peso mínimo de G' con las aristas marcadas en el paso 1 del algoritmo cubren todos los vértices de G y el grafo resultante no contiene ciclos. Además, como el árbol generador de peso mínimo de G' es también inducido por el algoritmo de Boruvka,

es un árbol generador de peso mínimo de G' es un subgrafo del árbol generador de peso mínimo de G. Por tanto, la unión de las aristas escogidas durante el apso 2 del algoritmo de Boruvka con las aristas del árbol generador de peso mínimo de G' es el árbol generador de peso mínimo de G. □

En nuestro repositorio [7] encontramos implementado este algoritmo. A modo de ejemplo, lo hemos ejecutado sobre el grafo de la Figura 6.20.

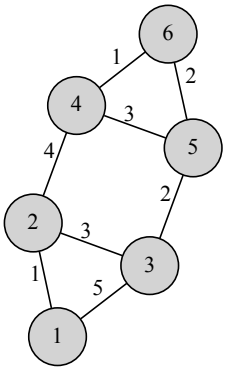


Figura 6.20

Las órdenes para su realización son:

```
from Boruvka import *

g=Grafo([],[(1,2),(1,3),(2,3),(2,4),(3,5),(4,5),(4,6),(5,6)])

g.ponderado([1,5,3,4,2,3,1,2])

Boruvka(g, True)
```

Y la salida obtenida por pantalla la encontramos en la Figura 6.21.

6.1. Caminos de peso mínimo

Expondremos ahora tres algoritmos que nos permitirán encontrar, dentro de un grafo no dirigido, el camino de peso mínimo entre dos vértices cualesquiera. Es importante abordar esta situación ya que, en ocasiones, existen varios árboles de peso mínimo para un mismo grafo, por lo que la elección de uno u otro puede privarnos de la obtención de las rutas óptimas entre dos de sus nodos.

La búsqueda del camino de peso mínimo tiene aplicaciones reales muy diversas e incluso de utilidad cotidiana como el cálculo de rutas óptimas dado un punto de partida y un destino.

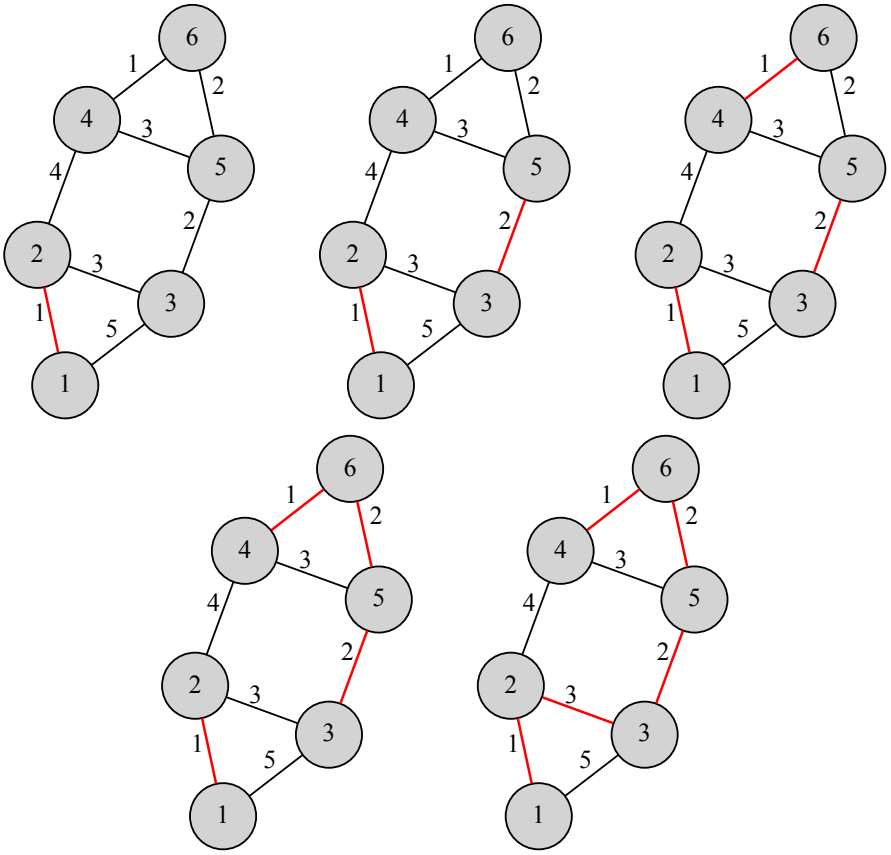


Figura 6.21. Algoritmo de Boruvka.

A.14 Algoritmo de Dijkstra

Este algoritmo consiste en buscar el camino de menor peso entre dos vértices. Para ello, se parte del vértice inicial y se van comparando los pesos de los posibles caminos considerando los vértices intermedios. Denotemos entonces v_i como vértice inicial, v_f como vértice final y v_a como vértice actual.

1. Se parte de v_i, es decir $v_a = v_i$ y se crea un vector distancias D donde el elemento D_j indica la distancia de v_i al nodo v_j. Estas distancias, antes de comenzar el algoritmo, se tomarán como ∞ para aquellos vértices no adyacentes a v_i. Si v_j es adyacente a v_i, D_j contiene el peso de la arista que los une. Se elige, de entre los vértices adyacentes a v_a, la arista de menor peso y se toma v_a como el extremo final de la arista elegida. Se marcará entonces v_a para indicar que ya tenemos el camino de peso mínimo desde v_i hasta él.

2. Mientras no se alcance v_f, se calculan las distancias desde v_i hasta los nodos adyacentes a v_a. Si alguna de esas distancias desde v_i hasta v_j, siendo v_j vecino de v_a, es menor que la distancia acumulada en D_j, se actualiza D_j con este valor menor. Se marca el mínimo de los elementos no marcados de D y se toma el vértice v_j como v_a.

3. El proceso termina cuando marcamos v_f, pues esto indica que hemos encontrado el camino de peso mínimo hasta él.

Es decir, el algoritmo de Dijkstra, en cada iteración, aumenta la longitud de los caminos que estudia, recorre todos los caminos posibles entre v_i y los vértices adyacentes al nodo actual y, si el peso de un camino nuevo que no contiene alguno de los vértices ya seleccionados es el menor, se actualizan los vértices y aristas escogidas.

Veamos el funcionamiento del algoritmo de Dijkstra, apliquémoslo para encontrar el camino de peso mínimo entre el nodo 1 y 4 en el grafo de la Figura 6.22.

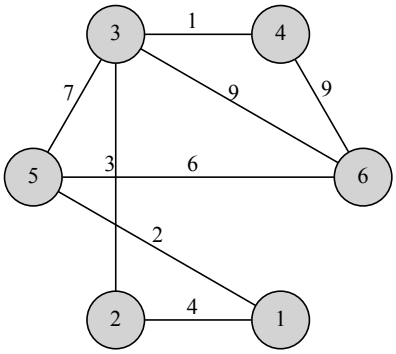


Figura 6.22

El punto de partida del algoritmo son dos vectores D y C. El vector de D es el vector distancias, donde cada elemento indica la distancia entre el vértice inicial y el resto de los vértices (ordenados) del grafo (notaremos por infinito la distancia entre dos vértices no conectados por un camino). El vector C lo utilizaremos para indicar la ruta que seguimos desde v_i hasta el resto de los vértices en el camino de peso mínimo. Cada elemento del vector C equivale a un vértice del grafo.

Comencemos definiendo D y C en nuestro grafo de partida, tomemos como vértice inicial $v_i = 1$ y como vértice final $v_f = 4$. En D, para la primera iteración, se consideran los caminos de longitud 1 entre v_i y el resto de los vértices.

- $D = (0, 4, \infty, \infty, 2, \infty)$,
- $C = (1, 1, 1, 1, 1, 1)$.

Es decir, D me indica que, por ejemplo, la distancia entre el nodo 1 y él mismo es 0, como es lógico, que la distancia entre los nodos 1 y 2 es 4, dado que es el peso de la arista que los une y que la distancia entre los nodos 1 y 3 es infinita, dado que no son adyacentes, pues recordemos que, al estar en la primera iteración, se estudian los caminos de longitud 1. Por otro lado, C es un vector de unos pues estamos en la primera iteración del algoritmo, es decir, hemos considerado la distancia a todos los vértices tomando como camino únicamente el

vértice 1. El vector C nos dice ahora mismo que las distancias indicadas en D se han calculado considerando que estamos en el nodo 1.

El siguiente paso consiste en elegir, dentro de D, el mínimo (exceptuando el valor 0 de la distancia entre v_i con él mismo), en nuestro caso, el valor 2 (si hubiera más de un valor mínimo se escoge uno al azar). El valor 2 es el peso de la arista (1,5), con lo cuál, marcamos D_5 y el nodo 5 para indicar que ya hemos encontrado el camino mínimo entre v_i y 5. Además, actualizamos los vectores D y C considerando ahora el nodo 5 como vértice intermedio, es decir, algunos vértices que antes no eran alcanzables desde 1, ahora lo podrán ser a través de 5.

Así por ejemplo, vemos que podemos llegar al nodo 6. La distancia entre 1 y 6 será la distancia de 1 a 5 más la distancia de 5 a 6, es decir $2+6=8$. Como 8 es menor que infinito, que era la distancia que teníamos previamente entre 1 y 6, actualizamos en D el valor obtenido y actualizamos en C el nodo por el que accedíamos a 6, antes teníamos el nodo 1, ahora llegamos a través de 5.

- $D = (0, 4, \infty, \infty, \mathbf{2}, 8)$,
- $C = (1, 1, 1, 1, 1, 5)$.

Vemos que hemos marcado el elemento D_5 y hemos actualizado el valor de D_6. Además, hemos actualizado también el valor C_6, puesto que el nodo por el que llegamos al vértice 6 es el nodo 5.

Desde el nodo 5 podemos acceder también al nodo 3. La distancia desde v_i hasta él, pasando por 5, es 9, que es menor que la que teníamos en D_3 actualmente, infinito. Por tanto, actualizamos el valor D_3 y el valor C_3.

El resultado de esta segunda iteración es:

- $D = (0, 4, 9, \infty, \mathbf{2}, 8)$,
- $C = (1, 1, 5, 1, 1, 5)$.

El siguiente paso para comenzar la tercera iteración, es marcar la menor distancia en D sin considerar las ya marcadas ni el valor 0 de la distancia de v_i a v_i. En este caso el mínimo es $D_2 = 4$, por lo que marcaremos el vértice 2 y consideraremos ahora sus nodos vecinos para estudiar los caminos entre v_i y el resto de vértices tomando el nodo 2 como intermedio. Realizando el mismo procedimiento que en la anterior iteración, obtenemos:

- $D = (0, \mathbf{4}, 7, \infty, \mathbf{2}, 8)$,
- $C = (1, 1, 2, 1, 1, 5)$.

En esta iteración, he marcado el elemento D_2 y he actualizado los valores de D_3 y C_3, ya que la distancia entre 1 y 3 pasando por 2, 7, es menor que la que teníamos pasando por 5, 9. Por tanto, en C_3, el nodo por el que accedo al vértice 3 es ahora el nodo 2.

Para la cuarta iteración, marcamos el menor valor de D que no sea el 0 de la distancia de v_i a sí mismo ni los ya marcados. Marcamos entonces $D_3 = 7$. Por tanto, nos colocamos ahora en el vértice 3 y cotejamos las distancias entre el nodo 1 y los vecinos del nodo 3. Ahora, sabemos por C que para llegar al nodo 3 desde 1, debemos pasar por 2. Por tanto, estamos estudiando los caminos $1-2-3-v$ siendo v un vértice vecino al nodo 3.

Así, el resultado de la iteración es:

- $D = (0, \mathbf{4}, \mathbf{7}, 8, \mathbf{2}, 8)$,
- $C = (1,1,2,3,1,5)$.

Donde hemos actualizado la distancia que teníamos de 1 a 4, infinito, por la nueva calculada a través del camino $1-2-3-4$, que es 8. Por ello, actualizamos el valor C_4, indicando que llegamos al nodo 4 desde el vértice 3. También llegamos a los nodos 5 y 6 a través del vértice 3, pero las distancias por estos camino no mejora la acumulada en D, por lo que lo dejamos como estaba.

En el siguiente paso del algoritmo nos quedan dos valores iguales en D sin marcar. Al no haber aristas de peso negativo ni de peso 0, es claro que ninguna de ellas se podrá mejorar. Por tanto, ya podríamos marcar tanto D_4 como D_6. En particular, hemos obtenido la distancia mínima entre v_i y v_f, por lo que damos por concluido el proceso. Además, comprobamos que el algoritmo de Dijkstra nos ha dado no solo el camino de peso mínimo entre 1 y 4, Figura 6.23, sino los caminos de peso mínimo entre el nodo 1 y cualquiera de los demás vértices.

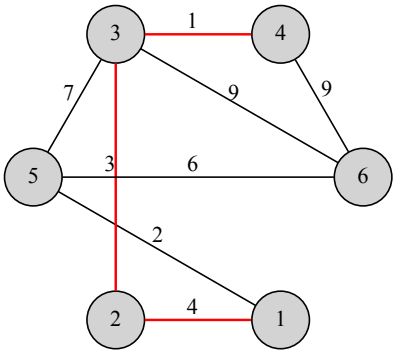


Figura 6.23

En nuestro repositorio [7] encontramos implementado este algoritmo. A modo de ejemplo, lo hemos ejecutado sobre el grafo de la Figura 6.24.

El código en este caso es:

```
from Dijkstra import *

g=Grafo([],[(1,2),(1,3),(2,3),(2,4),(3,5)])

g.ponderado([1,5,3,4,2])
```

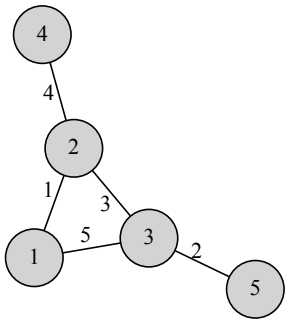


Figura 6.24

```
for i in g.vertices:
    print('Vértice inicial: ', str(i), '\n ')
    print(Dijkstra(g,i))
    print('\n')
```

La salida por pantalla obtenida es la siguiente:

```
Vértice inicial:  1

m=  [0, 1, 5, inf, inf]
m=  [0, 1, 4, 5, inf]
m=  [0, 1, 4, 5, 6]
m=  [0, 1, 4, 5, 6]
{'Pesos': [0, 1, 4, 5, 6], 'Ruta': [1, 1, 2, 2, 3]}


Vértice inicial:  2

m=  [1, 0, 3, 4, inf]
m=  [1, 0, 3, 4, inf]
m=  [1, 0, 3, 4, 5]
m=  [1, 0, 3, 4, 5]
{'Pesos': [1, 0, 3, 4, 5], 'Ruta': [2, 2, 2, 2, 3]}


Vértice inicial:  3

m=  [5, 3, 0, inf, 2]
m=  [5, 3, 0, inf, 2]
m=  [4, 3, 0, 7, 2]
m=  [4, 3, 0, 7, 2]
{'Pesos': [4, 3, 0, 7, 2], 'Ruta': [2, 3, 3, 2, 3]}
```

```
Vértice inicial:  4

m=  [inf, 4, inf, 0, inf]
m=  [5, 4, 7, 0, inf]
m=  [5, 4, 7, 0, inf]
m=  [5, 4, 7, 0, 9]
{'Pesos': [5, 4, 7, 0, 9], 'Ruta': [2, 4, 2, 4, 3]}


Vértice inicial:  5

m=  [inf, inf, 2, inf, 0]
m=  [7, 5, 2, inf, 0]
m=  [6, 5, 2, 9, 0]
m=  [6, 5, 2, 9, 0]
{'Pesos': [6, 5, 2, 9, 0], 'Ruta': [2, 3, 5, 2, 5]}
```

La correctitud del algoritmo de Dijkstra [1] queda probada por el siguiente resultado.

Proposición 21. *Dado un grafo ponderado G de pesos no negativos y establecido el vértice inicial s, el algoritmo de Dijkstra encuentra el camino de peso mínimo desde s a cualquier nodo de G.*

Demostración. Para esta demostración, utilizaremos la siguiente notación:

- Si $G = (V, E)$ es el grafo estudiado y $s \in V$ es el nodo inicial, para cada $v \in V$, denotaremos por $\delta(v)$ la distancia mínima desde s hasta v y por $\mathrm{d}(v)$ la distancia desde s hasta v que nos devuelve el algoritmo.
- El conjunto R será aquel que va incluyendo los vértices para los cuáles encontramos el camino de menor peso a través del algoritmo. Una vez marcamos el vértice v, hacemos $R = R \cup \{v\}$.
- Dados $u, v \in V$, denotaremos por $\mathrm{l}(uv)$ el peso de la arista (u, v).

Así pues, realicemos esta demostración por inducción sobre R. Queremos probar que el vector de distancias que nos devuelve el algoritmo contiene las distancias mínimas desde s a cualquier vértice.

Si R contiene un solo elemento, entonces debe ser $R = \{s\}$, por lo que $\mathrm{d}(s) = 0$, que es la distancia mínima.

Sea ahora u el nodo que hemos establecido como vértice final, u será entonces el último nodo añadido a R. Sea entonces $R = R' \cup \{u\}$. Supondremos como hipótesis de inducción que $\mathrm{d}(x) = \delta(x)$ para todo $x \in R'$. Por tanto, solo falta demostrar que $\mathrm{d}(u) = \delta(u)$.

Supongamos, para llegar a contradicción, que el camino de peso mínimo entre s y u es Q y sea $\mathrm{l}(Q)$ la longitud de dicho camino. Tenemos entonces que

$$\mathrm{l}(Q) \geq \mathrm{d}(u).$$

Sabemos que el camino Q comienza en algún vértice contenido en R' pero, al ser u su destino, como $u \notin R$, llegará un momento en que Q visite un vértice que no está en R'. Sea (x, y) la arista por la que Q abandona R' y sea Q_x un subcamino dentro de Q que llega hasta x. Es claro entonces que

$$\mathrm{l}(Q_x) + \mathrm{l}(xy) \leq \mathrm{l}(Q).$$

Por hipótesis, como $x \in R'$, tenemos que el camino mínimo de s a x es $\mathrm{d}(x)$, por tanto, se tiene que

$$\mathrm{d}(x) \leq \mathrm{l}(Q_x) \rightarrow \mathrm{d}(x) + \mathrm{l}(xy) \leq \mathrm{l}(Q).$$

Como y es adyacente a x y el algoritmo estudia los vecinos de los nodos alcanzables por s, como x en este caso, podemos afirmar que y ha sido analizado en el algoritmo, por tanto $\mathrm{d}(y)$ es la menor distancia desde s hasta y. Así,

$$\mathrm{d}(y) \leq \mathrm{d}(x) + \mathrm{l}(xy).$$

Finalmente, cuanto el algoritmo estudia u, se obtiene $d(u)$ como distancia mínima desde s a u, entonces $\mathrm{d}(u) \leq \mathrm{d}(y)$. Combinando ahora las desigualdades obtenidas, obtenemos que $\mathrm{d}(x) < \mathrm{d}(x)$, es decir, una contradicción. Por tanto, no existe tal camino Q de menor peso que $\mathrm{d}(u)$, por tanto, $\mathrm{d}(u) = \delta(u)$, como queríamos. □

A continuación, vamos a proponer dos alternativas al algoritmo de Dijkstra, ya que este pone condiciones sobre las aristas, que sean no negativas. Estudiaremos ahora dos métodos los cuáles, a pesar de requerir mayor tiempo de ejecución que el algoritmo de Dijkstra, permiten encontrar el camino de peso mínimo en grafos cuyas aristas pueden tener pesos negativos, además de que nos permiten detectar si en dicho grafo existen ciclos negativos, es decir, ciclos donde la suma total de los pesos de las aristas que lo forman es negativa.

Es importante saber diferenciar los casos que podemos encontrarnos. En el caso de los grafos dirigidos, si existen aristas de pesos negativos pero no existen ciclos negativos, sí es posible encontrar el camino de peso mínimo entre dos vértices cualesquiera. Sin embargo, si nuestro grafo contiene un ciclo negativo, no podemos encontrar dicho camino, ya que, conforme se recorre, se reduce aún más el coste o peso. En el caso de grafos no dirigidos, en ninguno de los casos podemos asegurar que encontraremos el camino de peso mínimo para dos vértices cualesquiera pues, solo con la existencia de una arista de peso negativo, al recorrerla en ambos sentidos, estamos formando ya un ciclo negativo.

A.15 Algoritmo de Bellman-Ford

El funcionamiento de este algoritmo es muy similar al de Dijkstra. En este caso, debemos relajar cada arista $n-1$ veces, siendo n el orden del grafo.

En este tipo de algoritmos se estudian caminos distintos entre vértices iguales para escoger el de peso mínimo. Por relajar una arista entendemos que, en la búsqueda de dicho camino, he escogido una arista que me proporciona un camino de menor peso que el que tengo actualmente.

1. Inicializamos una variable "iteraciones" a 0, escogemos nuestro vértice inicial v_i y creamos la matriz de distancias D cuyos elementos serán los pesos de las aristas que unen v_i con cada vértice, siendo estos elementos ∞ si no hay arista comunicando v_i con dicho nodo.

2. Para cada uno de los vértices del grafo:

   - Se toma dicho vértice como vértice actual v_a.
   - Se toman los vecinos de v_a.
   - Para cada uno de los vecinos v de v_a se calcula la distancia $d(v_i, v_a) + d(v_a, v)$.
   - Si dicha distancia es menor que $d(v_i, v)$, se cambia en D el elemento D_v.

3. Se incrementa en 1 la variable iteraciones. Si su valor es $n-1$, siendo n el orden del grafo, se finaliza el proceso. En caso contrario, volvemos al paso dos con el grafo resultante.

   Para comprobar si existe un ciclo negativo en el grafo, simplemente debemos realizar una iteración más una vez haya finalizado el algoritmo. Si ha sido posible relajar alguna arista, entonces existe un ciclo negativo en dicho grafo.

Visualicemos el comportamiento de este algoritmo aplicándolo sobre el grafo de la Figura 6.25. En este caso, hagamos el peso de la arista $(3,4)$ igual a -21, para observar el funcionamiento con un ciclo negativo, sea dicho ciclo $c = [3,6,4,3]$, cuya suma total de los pesos de las aristas que lo componen es -3.

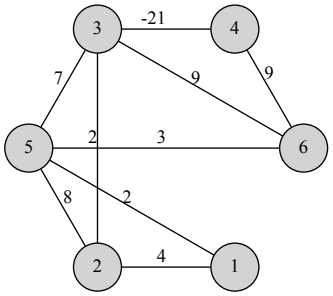
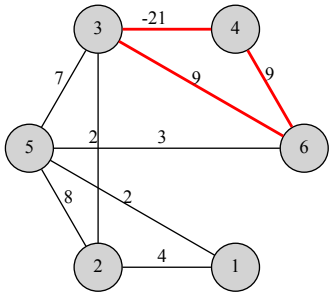


Figura 6.25



Tomemos, de nuevo, como vértice inicial $v_i = 1$. Inicializamos el vector distancias $D = (0,4,\infty,\infty,2,\infty)$ y el vector de ruta $C = (1,1,1,1,1,1)$. Recordemos que cada elemento D_k es la distancia desde v_i hasta v_k y cada elemento C_k es el vértice inmediatamente anterior por el que se accede a v_k en el camino mínimo actual.

Comenzamos la primera iteración, partiendo de $v_i = 1$ solo podemos acceder a los nodos 2 y 5. La distancia a estos nodos desde v_1 ya la tenemos incluida en la inicialización de D. Por tanto, calculamos ahora los caminos desde v_i a todos los vértices vecinos de los nodos 2 y 5.

Empezamos con los nodos vecinos a 2: 3 y 5.

- $d(1,2) + d(2,3) = 4 + 2 = 6 < d(1,3) = D_3 = \infty$; hacemos $D_3 = 6$ y $C_3 = 2$ (el nodo anterior a 3 desde el que accedemos a él).
- $d(1,2) + d(2,5) = 4 + 8 = 12 > d(1,5) = D_5 = 2$; no actualizamos los valores D_5 y C_5.

Estudiamos ahora los vértices vecinos a 5: 2,3 y 6.

- $d(1,5) + d(5,2) = 2 + 8 = 10 > d(1,2) = 4$; no actualizamos ningún valor.
- $d(1,5) + d(5,3) = 2 + 7 = 9 > d(1,3) = 6$; no actualizamos valores.
- $d(1,5) + d(5,6) = 2 + 3 = 5 < d(1,6) = \infty$; hacemos $D_6 = 5$ y $C_6 = 5$.

Obtenemos entonces los vectores:

- $D = (0,4,6,\infty,2,5)$,
- $C = (1,1,2,1,1,5)$.

Siguiendo todavía en la primera iteración, estudiamos ahora las distancias entre v_i y los vértices vecinos a 3 y 6.

Estudiemos los vértices vecinos al nodo 3: 2, 4, 5 y 6.

- $d(1,3) + d(3,2) = 6 + 2 = 8 > d(1,2) = 4$; no actualizamos ningún valor.
- $d(1,3) + d(1,4) = 6 + (-21) = -15 < d(1,4) = \infty$; hacemos $D_4 = -15$ y $C_4 = 3$

- $d(1,3)+d(3,5)=6+7=13>d(1,5)=2$; no actualizamos ningún valor de D ni de C.
- $d(1,3)+d(3,6)=6+9=15>d(1,6)=5$; no actualizamos ningún valor.

Vamos ahora con los nodos vecinos al vértice 6: 3, 4 y 5.

- $d(1,6)+d(6,3)=5+9=14>d(1,3)=6$; no hacemos ningún cambio en los vectores D y C.
- $d(1,6)+d(6,4)=5+9=14>d(1,4)=-15$; no actualizamos ningún valor.
- $d(1,6)+d(6,5)=5+3=8>2$; no modificamos ningún elemento de D ni de C.

Finalmente, para terminar esta primera iteración, llegamos al vértice 4, que tiene como nodos vecinos a 3 y 6.

- $d(1,4)+d(4,3)=-15+(-21)=-37<d(1,3)=6$; hacemos $D_3=-37$ y $C_3=4$.
- $d(1,4)+d(4,6)=-15+9=-6<5$; actualizamos los valores $D_6=-6$ y $C_6=4$.

De esta forma, tras la primera iteración del algoritmo, obtendríamos los vectores:

- $D=(0,4,-37,-15,2,-6)$,
- $C=(1,1,4,3,1,4)$.

Para la segunda iteración procedemos de forma similar. En este caso, actualizamos los siguientes valores:

- $d(1,3)+d(3,2)=-37+2=-35<d(1,2)=4$; $D_2=-35, C_2=3$.
- $d(1,3)+d(3,4)=-37-21=-58<d(1,4)=-15$; $D_4=-58$.
- $d(1,3)+d(3,5)=-37+7=-30<d(1,5)=2$; $D_5=-30, C_5=3$.
- $d(1,3)+d(3,6)=-37+9=-26<d(1,6)=-6$; $D_6=-26, C_6=3$.
- $d(1,4)+d(4,3)=-58-21=-79<d(1,3)=-37$; $D_3=-79, C_3=4$.
- $d(1,4)+d(4,6)=-58+9=-47<d(1,6)=-26$; $D_6=-47, C_6=4$.

Así, los vectores obtenidos en esta segunda iteración son:

- $D=(0,-35,-79,-58,-30,-47)$,
- $C=(1,3,4,3,3,4)$.

Es fácil comprobar que en las 3 iteraciones que restan para finalizar el algoritmo siempre realizarán cambios en las distancias, es decir, las reducirán.

Por este motivo, una vez finalizado el algoritmo, al realizar una última iteración, modificaremos algún elemento de D, concluyendo por tanto la existencia de ciclos negativos en nuestro grafo, como ya sabíamos.

La validez del algoritmo de Bellman-Ford [14] queda demostrada con el siguiente resultado.

Lema 6. *Sea $G = (V, E)$ un grafo sobre el que se ha aplicado el algoritmo de Bellman-Ford, tomando como vértice inicial $s \in V$ y notemos por $\mathrm{d}(u)$ el peso del camino que une s con un vértice $u \in V$. Tras n iteraciones del algoritmo, si $\mathrm{d}(u) \neq \infty$, entonces es igual al peso de algún camino desde s hasta u y, si hay un camino de s a u con, a lo sumo, n aristas, entonces $\mathrm{d}(u)$ es, a lo sumo, la longitud del camino más corto de s a u de, como máximo, n lados.*

Demostración. Realicemos la demostración por inducción sobre el número de iteraciones del algoritmo.

Sea $n = 0$ y consideremos el momento antes de ejecutarse el algoritmo por primera vez. Tenemos que $\mathrm{d}(s) = 0$, por lo que se verifica el resultado. Además, $\mathrm{d}(u) = \infty$ para todo $u \in V$, ya que no hay ningún camino de 0 aristas que conecte s con cualquier otro vértice distinto a él.

Para el caso inductivo, probemos en primer lugar la primera parte. considérese un momento en que la distancia de un vértice v se actualiza como sigue: $\mathrm{d}(v) = \mathrm{d}(u) + \mathrm{w}(u, v)$, siendo $\mathrm{w}(u, v)$ el peso de la arista (u, v). Por hipótesis de inducción, $\mathrm{d}(u)$ es el peso de algún camino entre s y u, entonces $\mathrm{d}(u) + \mathrm{w}(u, v)$ es el peso de un camino de s a v pasando por u.

Para la segunda parte, supongamos el camino de peso mínimo P entre s y u con un máximo de n aristas. Sea v el último vértice de dicho camino antes de llegar a u en P. Entonces, el camino desde s hasta v en P es el camino de peso mínimo con, a lo sumo, $n-1$ aristas entre ellos pues, de no ser así, habría un camino de peso estrictamente menor de s a v con no mas de n lados y luego podríamos añadirle la arista (u, v) para obtener el camino con un máximo de n lados de peso estrictamente menor que P, llegando así a una contradicción.

Por hipótesis de inducción, $\mathrm{d}(v)$, tras $n-1$ iteraciones del algoritmo, es, como máximo, el peso de este camino entre s y v. Por tanto, $\mathrm{w}(u, v) + \mathrm{d}(v)$ es, como máximo, la longitud de P. En la iteración n-ésima, $\mathrm{d}(u)$ se compara con $\mathrm{w}(u, v) + \mathrm{d}(v)$ y se hace igual a esa cantidad si es mayor que ella. Por tanto, tras n iteraciones $\mathrm{d}(u)$ es, como máximo, la longitud de P, es decir, el peso del camino de peso mímino entre s y u, con, a lo sumo, n aristas.

Con respecto a la existencia de ciclos no negativos, si los hay, cada camino de peso mínimo visita una sola vez a cada vértice, como máximo, luego, una iteración más no hará mejoras en el resultado. Recíprocamente, supongamos que los resultados no se pueden mejorar, entonces para cada ciclo $[v_0, v_1, \ldots, v_{k-1}]$ se tiene que:

$$\mathrm{d}(v_i) \leq \mathrm{d}(v_{i-1 mod k}) + \mathrm{w}(v_{i-1 mod k}, v_i).$$

Sumando alrededor del ciclo $d(v_i)$ y $d(v_{i-1 mod k})$, los términos de distancia se cancelan y queda:

$$0 \leq \sum_{i=1}^{k} w(v_{i-1 mod k}, v_i).$$

Es decir, cada ciclo tiene peso no negativo. □

En nuestro repositorio [7] hemos implementado el algoritmo de Bellman-Ford. A modo de ejemplo, lo hemos aplicado sobre el grafo de la Figura 6.26.

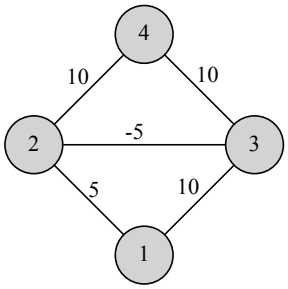


Figura 6.26

El código a ejecutar es el siguiente:

```
from Bellman_Ford import *

g=Grafo([],[(1,2),(1,3),(2,3),(2,4),(3,4)])

g.ponderado([5,10,-5,10,10])

for i in g.vertices:
    print('Vértice ' + str(i) + ': ')
    b=Bellman_Ford(g,i)
    print('Output:', b, '\n')
```

Obteniéndose la siguiente salida por pantalla:

```
Vértice 1:
[0, 5, 10, inf]
[0, -5, 0, 10]
[0, -15, -10, 0]
[-10, -25, -20, -10]
Hay ciclo negativo
Output: [[-20, -35, -30, -20], [2, 3, 2, 3]]

Vértice 2:
[5, 0, -5, 10]
[5, -10, -5, 5]
```

```
[-5, -20, -15, -5]
[-15, -30, -25, -15]
Hay ciclo negativo
Output: [[-25, -40, -35, -25], [2, 3, 2, 3]]

Vértice 3:
[10, -5, 0, 10]
[0, -15, -10, 0]
[-10, -25, -20, -10]
[-20, -35, -30, -20]
Hay ciclo negativo
Output: [[-30, -45, -40, -30], [2, 3, 2, 3]]

Vértice 4:
[inf, 10, 10, 0]
[15, 0, 5, 0]
[5, -10, -5, 0]
[-5, -20, -15, -5]
Hay ciclo negativo
Output: [[-15, -30, -25, -15], [2, 3, 2, 3]]
```

A.16 Algoritmo de Floyd-Warshall

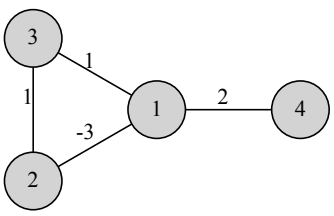
Este algoritmo nos sirve también para la detección de ciclos negativos. El procedimiento constará, como máximo, de tantas iteraciones como vértices tenga el grafo.

1. Partimos de la matriz de distancias inicial D, donde $D_{ij} = p$, siendo p el peso de la arista (i, j). En caso de no existir la arista (i, j), haremos $p = \infty$.

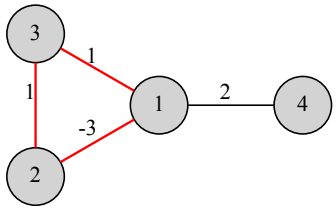
2. Para cada vértice v del grafo, comprobamos si la distancia D_{ij} es mayor que el valor $a = D_{iv} + D_{vj}$, en tal caso, actualizamos el valor D_{ij} por a. Comprobamos si algún elemento de la diagonal de D es negativo, en tal caso, habrá un ciclo negativo y terminaríamos el proceso.

3. En caso de tener elementos no negativos en la diagonal de D, para comprobar si hay ciclos negativos, basta con realizar una iteración más del algoritmo y ver si se ha modificado algún valor de D, en tal caso, habrá un ciclo negativo.

Veamos trabajar este algoritmo sobre el siguiente grafo,



que observamos que posee un ciclo negativo:



Partimos entonces de la matriz de distancias:

$$\mathbf{D} = \begin{pmatrix} 0 & -3 & 1 & 2 \\ -3 & 0 & 1 & \infty \\ 1 & 1 & 0 & \infty \\ 2 & \infty & \infty & 0 \end{pmatrix}.$$

Tomamos ahora como nodo intermedio el vértice 1.

- $d(2,1) = -3 = d(2,1) + d(1,1) = -3 + 0 = -3$; no cambiamos nada.
- $d(2,2) = 0 > d(2,1) + d(1,2) = -3 - 3 = -6$; hacemos $D_{22} = -6$.
- $d(2,3) = 1 > d(2,1) + d(1,3) = -3 + 1 = -2$; hacemos $D_{23} = D_{32} = -2$.
- $d(2,4) = \infty > d(2,1) + d(1,4) = -6 + 2 = -4$; hacemos $D_{24} = D_{42} = -4$.
- $d(3,1) = 1 = d(3,1) + d(1,1) = 1 + 0 = 1$; no cambiamos nada.
- $d(3,2) = -2 = d(3,1) + d(1,2) = 1 - 3 = -2$; no cambiamos nada.
- $d(3,3) = 0 < d(3,1) + d(1,3) = 1 + 1 = 2$; no cambiamos nada.
- $d(3,4) = \infty > d(3,1) + d(1,4) = 1 + 2 = 3$; $D_{34} = D_{43} = 3$.
- $d(4,1) = 2 = d(4,1) + d(1,1) = 2 + 0 = 2$; no cambio nada.
- $d(4,2) = -1 = d(4,1) + d(1,2) = 2 - 3 = -1$; no cambio nada.
- $d(4,3) = 3 = d(4,1) + d(1,3) = 2 + 1 = 3$; no cambio nada.
- $d(4,4) = 0 < d(4,1) + d(1,4) = 2 + 2 = 4$; no cambio nada.

Concluimos así la primera iteración, quedando la matriz de distancias como sigue:

$$\mathbf{D} = \begin{pmatrix} 0 & -3 & 1 & 2 \\ -3 & -6 & -2 & -1 \\ 1 & -2 & 0 & 3 \\ 2 & -1 & 3 & 0 \end{pmatrix}.$$

Comprobamos que, en la matriz resultante, el elemento D_{22} es negativo, por lo que podemos concluir que habrá ciclos negativos en este grafo y no podemos

calcular, en consecuencia, el camino de peso mínimo. Aún así, para continuar con el ejemplo, hagamos una iteración más. Para la segunda iteración, tomaremos el vértice 2 como nodo intermedio y repitiendo los cálculos de la iteración anterior, es fácil comprobar que D ha cambiado todos sus elementos:

$$\mathbf{D} = \begin{pmatrix} -6 & -15 & -29 & -28 \\ -15 & -12 & -26 & -25 \\ -29 & -26 & -52 & -51 \\ -28 & -25 & -51 & -50 \end{pmatrix}.$$

Es claro que en las dos iteraciones restantes los valores de D seguirán disminuyendo, en particular, al realizar una iteración más, D volverá a cambiar sus valores, además de tener valores negativos en su diagonal. Ambos sucesos nos permiten afirmar que existe algún ciclo negativo en el grafo estudiado, como ya sabíamos.

En nuestro repositorio [7] hemos realizado la implementación de este algoritmo. A modo de ejemplo, lo hemos ejecutado sobre el grafo de la Figura 6.27.

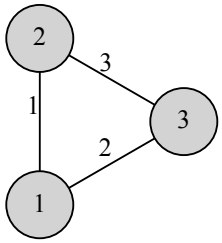


Figura 6.27

El código a ejecutar es el siguiente:

```
from Floyd_Warshall import *

g=Grafo([],[(1,2),(1,3),(2,3)])

g.ponderado([1,2,3])

Floyd_Warshall(g, True)
```

La salida de esta implementación la encontramos a continuación:

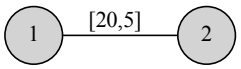
```
Inicio
[[0, 1, 2], [1, 0, 3], [2, 3, 0]]

Nodo intermedio: 1
[[0, 1, 2], [1, 0, 3], [2, 3, 0]]
```

```
Nodo intermedio: 2
[[0, 1, 2], [1, 0, 3], [2, 3, 0]]

Nodo intermedio: 3
[[0, 1, 2], [1, 0, 3], [2, 3, 0]]
```

Finalmente, para terminar esta sección, el último método que trataremos es el *Algoritmo de Ford-Fulkerson*, el cuál consiste en, dados el nodo inicial y final en un grafo ponderado, buscar caminos donde se pueda ampliar el flujo (peso de cada arista que conforma ese camino) hasta hacerlo máximo. La novedad de este algoritmo con respecto a los expuestos anteriormente es que, en lugar de tener un solo peso para cada arista, ahora tendremos dos, un peso que indica el flujo de salida y otro el flujo de retorno. Es decir:



Esta arista indica que, del vértice 1 al 2, hay un flujo de salida de 20 unidades y un flujo de retorno de 5 unidades. Análogamente para el vértice dos, el cuál tiene flujos de salida y retorno de 5 y 20 unidades, respectivamente.

Este algoritmo es muy importante debido a la aplicabilidad que tiene en la actualidad para cualquier proceso que implique el abastecimiento con cierto producto de un conjunto de elementos (receptores), desde la fuente de envío hasta el punto de llegada o sumidero. El objetivo es conseguir aprovechar al máximo el caudal de las vías de comunicación (canales) entre dichos puntos, pero evitando la congestión, es decir, hacer circular la máxima cantidad de producto sin retenciones.

A.17 Algoritmo de Ford-Fulkerson

1. Creamos un vector vacío $F = [\,]$. Tomamos como vértice actual, v_a, el vértice inicial, v_i y escogemos la arista incidente en v_a con mayor peso de salida, siempre que ésta no me devuelva a v_i.

2. Mientras $v_a \neq v_f$, siendo v_f el vértice final, vamos escogiendo la arista (v_a, u) de mayor peso de salida, con la condición de que no me devuelva al nodo inicial.

3. Se escoge el mínimo de todos los pesos de salida de las aristas elegidas, sea éste k, y se guarda en el vector F.

4. Para cada arista seleccionada, se reduce su peso de salida en k unidades y se aumenta su peso de retorno en k unidades.

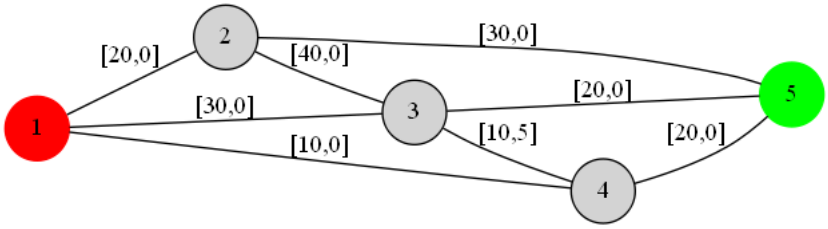
5. Si v_i ya no tiene aristas con pesos de salida positivo, vamos al paso 6, en caso contrario, volvemos al paso 1.

6. El flujo máximo que admite esta red o grafo es la suma de todos los elementos de F. El flujo máximo para cada arista es:

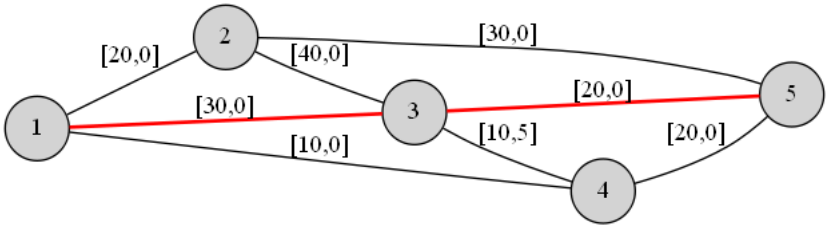
$$\max_{(i,j)\in E}\{p_{ij}^{inicial} - p_{ij}^{final}\}$$

siendo p_{ij} la dupla que contiene los pesos de salida y retorno de la arista (i,j).

Hagamos un ejemplo para entender mejor el funcionamiento de este método. Consideremos el siguiente grafo ponderado, donde hemos escogido como vértices inicial y final los nodos 1 y 5, respectivamente.



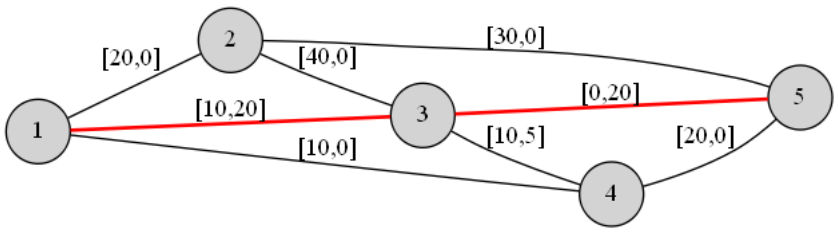
Creamos nuestra lista $F = [\,]$ donde guardaremos los distintos flujos k.
Primera iteración. Estamos en el nodo 1, escogemos la arista $(1,3)$ por ser la que tiene mayor peso de salida, 30, y nos colocamos en el nodo 3. Una vez aquí, escogemos la arista $(3,5)$ por ser la de mayor peso de salida, 20. Hemos llegado al nodo final, 5, por lo que paramos.



Ahora, cogemos $k = \text{mín}\{30,20\} = 20$, guardamos k en F y actualizamos los pesos de las aristas $(1,3)$ y $(3,5)$ como sigue:

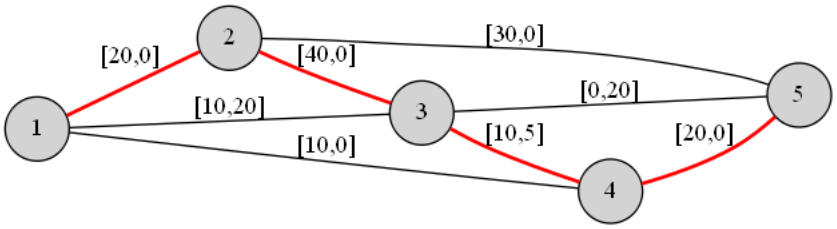
- $(p_{13}, p_{31}) = (30-k, 0+k) = (10,20)$
- $(p_{35}, p_{53}) = (20-k, 0+k) = (0,20)$

Obtenemos entonces el siguiente grafo:



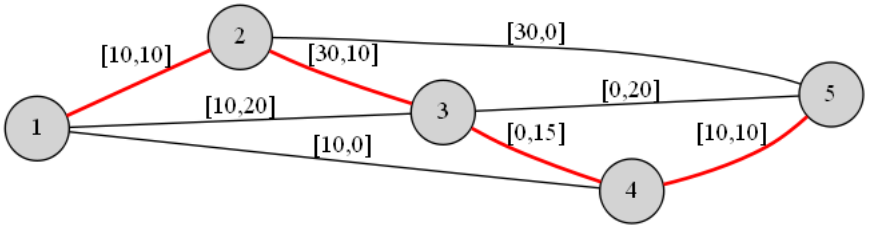
$F = [20]$

Segunda iteración. Ahora escogemos las aristas $(1,2)$, $(2,3)$ y... observamos que, si escogiésemos la arista incidente en 3 con mayor peso de salida, volveríamos a nuestro nodo inicial, por tanto, debemos elegir la arista incidente con el siguiente mayor peso de salida, en este caso, la $(3,4)$. Finalmente, escojo la arista $(4,5)$ y he llegado al nodo final.



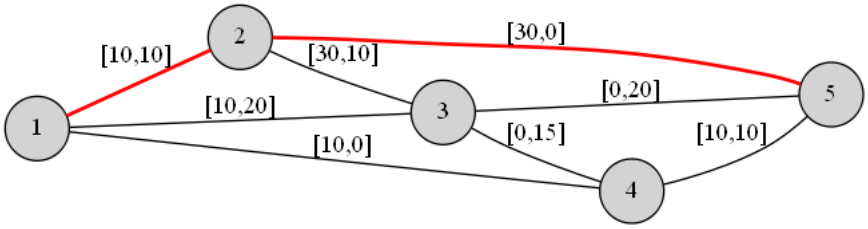
$k = \text{mín}\{20,40,10,20\} = 10$, añadimos k a F y actualizamos los pesos de las aristas escogidas:

- $(p_{12}, p_{21}) = (20-k, 0+k) = (10,10)$
- $(p_{23}, p_{32}) = (40-k, 0+k) = (30,10)$
- $(p_{34}, p_{43}) = (10-k, 5+k) = (0,15)$
- $(p_{45}, p_{54}) = (20-k, 0+k) = (10,10)$



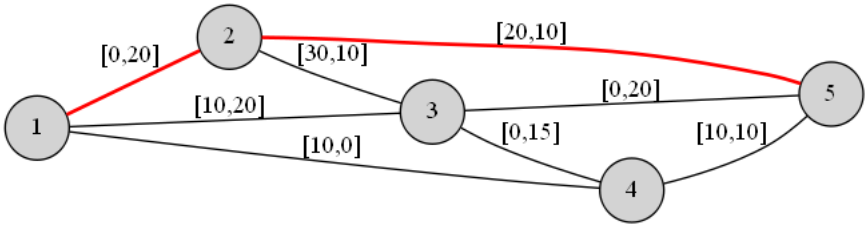
$F = [20,10]$

Tercera iteración. Ahora, todas las aristas incidentes en 1 tienen el mismo peso de salida, en este caso, elegimos cualquiera de ellas. Sea ésta, por ejemplo, la arista $(1,2)$. Ya sabemos que la $(2,3)$ no podemos elegirla, pues la única arista incidente en el vértice 3 con peso de salida positivo me devolvería a 1, por tanto, solo nos queda la $(2,5)$. Hemos llegado al nodo final.



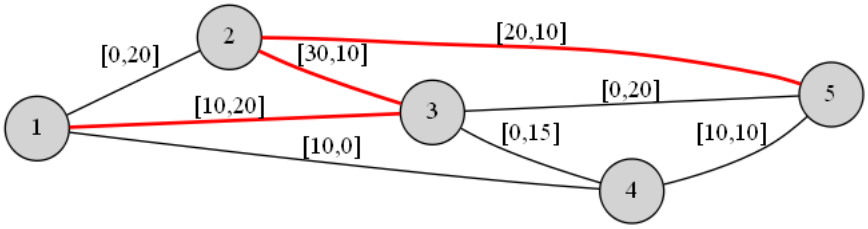
$k = \text{mín}\{10,30\} = 10$, añadimos k a F y actualizamos los pesos de las aristas escogidas:

- $(p_{12}, p_{21}) = (10-k, 10+k) = (0,20)$
- $(p_{25}, p_{52}) = (30-k, 0+k) = (20,10)$



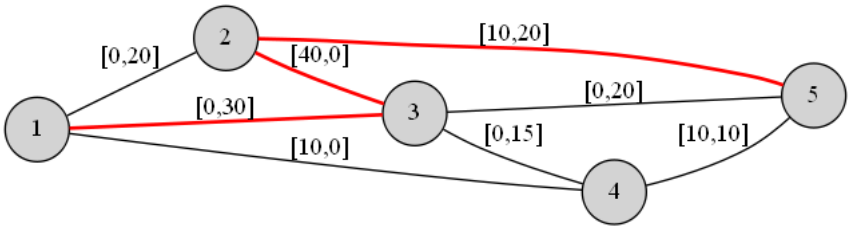
$F = [20,10,10]$

Cuarta iteración. Escogemos las aristas $(1,3)$, $(3,2)$ y $(2,5)$.



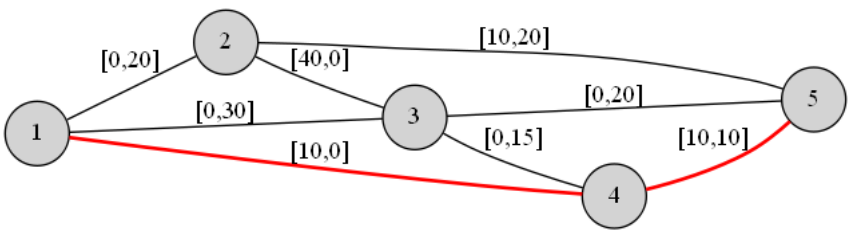
$k = \text{mín}\{10,10,20\} = 10$, añadimos k a F y actualizamos los pesos de las aristas escogidas:

- $(p_{13}, p_{31}) = (10-k, 20+k) = (0,30)$
- $(p_{32}, p_{23}) = (10-k, 30+k) = (0,40)$
- $(p_{25}, p_{52}) = (20-k, 10+k) = (10,20)$



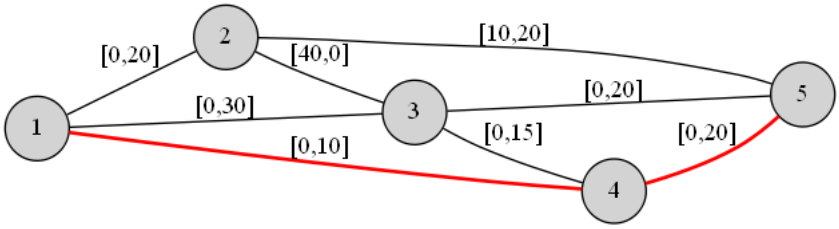
$F = [20,10,10,10]$

Quinta iteración. Escogemos las aristas $(1,4)$ y $(4,5)$. Notemos que, aunque deberíamos elegir la arista $(4,3)$, el vértice 3 nos llevaría al nodo 1, por tanto la descartamos.



$k = \text{mín}\{10, 10\} = 10$, añadimos k a F y actualizamos los pesos de las aristas escogidas:

- $(p_{14}, p_{41}) = (10-k, 0+k) = (0, 10)$
- $(p_{45}, p_{54}) = (10-k, 10+k) = (0, 20)$



$F = [20, 10, 10, 10, 10]$

Llegados a este punto, ya no tenemos aristas con pesos salientes positivos del nodo inicial, por tanto, hemos terminado el proceso.

El flujo óptimo que admite el grafo viene determinado por la suma de todos los elementos de F.

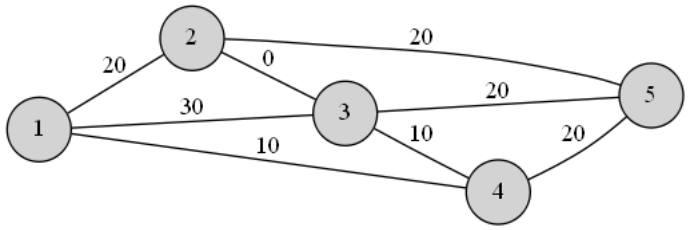
$$\sum_i F_i = 20 + 10 + 10 + 10 + 10 = 60.$$

Por otro lado, el flujo óptimmo que admite cada arista viene determinado por la diferencia entre los flujos iniciales y los finales, escogiendo de la dupla resultante, el valor máximo.

Así:

- $f_{12} : (20, 0) - (0, 20) = (20, -20)$; $f_{12} = 20$.
- $f_{13} : (30, 0) - (0, 30) = (30, -30)$; $f_{13} = 30$.
- $f_{14} : (10, 0) - (0, 10) = (10, -10)$; $f_{14} = 10$.
- $f_{23} : (40, 0) - (40, 0) = (0, 0)$; $f_{23} = 0$.
- $f_{25} : (30, 0) - (10, 20) = (20, -20)$; $f_{25} = 20$.
- $f_{34} : (10, 5) - (0, 15) = (10, -10)$; $f_{34} = 10$.
- $f_{35} : (20, 0) - (0, 20) = (20, -20)$; $f_{35} = 20$.
- $f_{45} : (20, 0) - (0, 20) = (20, -20)$; $f_{45} = 20$.

Así pues, el grafo de flujo máximo es:



$f_{max} = 60$

En nuestro repositorio [7] podemos encontrar una implementación del algoritmo de Ford-Fulkerson. Para ejemplificar su uso, lo hemos aplicado sobre el grafo de la Figura 6.28, donde hemos elegido el vértice 1 como nodo inicial y el vértice 5 como nodo final.

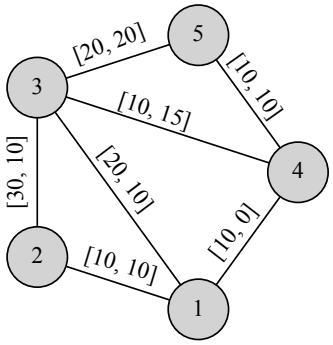


Figura 6.28

El código para su ejecución es:

```
from Ford_Fulkerson import *

g=Grafo([],[(1,2),(1,3),(1,4),(2,3),(3,4),(3,5),(4,5)])

g.ponderado([[10,10],[10,20],[10,0],[30,10],[10,15],\
                [20,20],[10,10]])

Ford_Fulkerson(g,1,5,True)
```

Y la salida por pantalla que obtendremos la encontramos en la Figura 6.29.

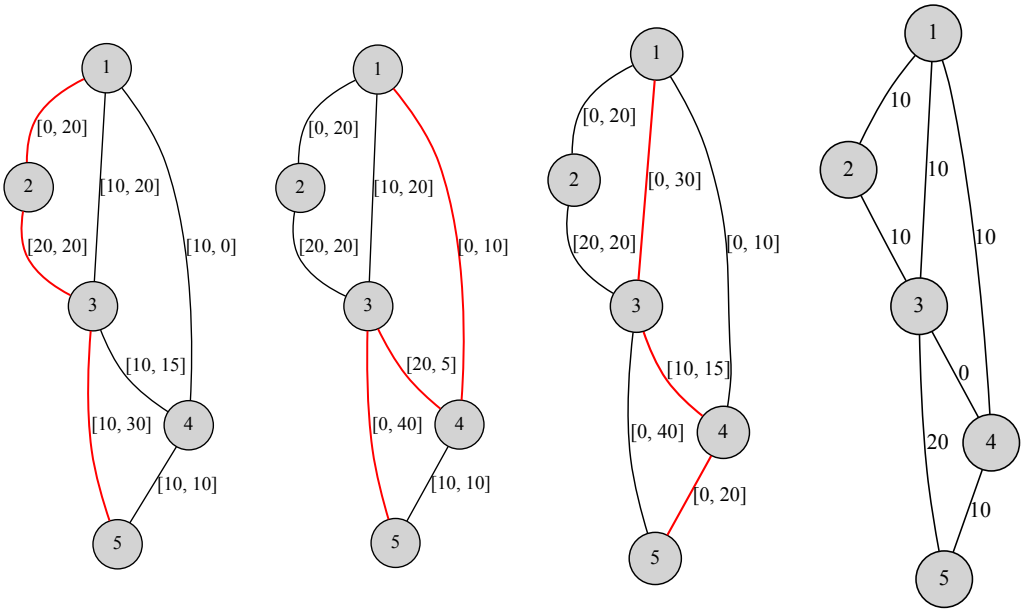


Figura 6.29. Algoritmo de Ford-Fulkerson.

Bibliografía

[1] Borradaile, G., Dijkstra's algorithm: Correctness by induction, disponible en https://web.engr.oregonstate.edu/~glencora/wiki/uploads/dijkstra-proof.pdf (visitado el 17 de febrero de 2019).

[2] Chiappa Raúl, Algunas motivaciones históricas en la teoría de grafos, Bahía Blanca, 1994, 37-52.

[3] Comellas F., Sánchez, A., Serra, O. y Fábrega, J., Matemática discreta, Edicions UPC, Universidad Politécnica de Cataluña, 2001.

[4] Conejero, A. y Jordán, C., Algoritmo de Hierholzer, Divulgación de las actividades científicas, tecnológicas y artísticas ocurridas en los tres campus de la UPV, disponible en http://www.upv.es/visor/media/777ac92e-82ea-2444-8ebc-1d275ca93d39/c (visitado el 16 de febrero de 2019).

[5] García Miranda, J., Lógica para informáticos (y otras herramientas matemáticas), Editorial Técnica Avicam, Granada, 2017.

[6] García sánchez, P. A., Álgebra-II, disponible en https://github.com/pedritomelenas/Algebra-II/tree/master/Grafos.

[7] García sánchez, P. A. y González Ibáñez, J., Lógica y Métodos Discretos, Algoritmos-sobre-grafos, disponible en https://github.com/lmd-ugr/Algoritmos-sobre-grafos.

[8] Gómez Saura, D., Reparto de costes en problemas de árbol generador de coste mínimo, Trabajo de fin de Máster, Universidad de Murcia, 2016.

[9] Liu, Y., Minimum Spanning Trees, accesible en http://community.wvu.edu/~krsubramani/courses/fa01/random/lecnotes/lec11/MST.pdf (visitado el 1 de febrero de 2019).

[10] Menéndez Velázquez, Amador, Una breve introducción a la teoría de grafos, SUMA, 28, (1998), 11-26.

[11] Rosen K. H., Michaels J. G., Gross J. L., Grossman J. W. and Shier D. R., Handbook of discrete and combinatorial mathematics, CRC Press, Boca Ratón, Londres, Washington D. C., 2000.

[12] Ruiz Recuenco, Francisco, Recorrido óptimo de los nodos de una red. Proyecto Fin de Carrera-Trabajo Fin de Grado, Facultad de Informática, UPM, 2009.

[13] Villarroel, Rafael, El teorema de Cayley, Matemáticas Discretas, Lima, UAEH, http://rvf0068.github.io/discretas/blog/2015/10/08/teorema-cayley/ (visitado el 20 de febrero de 2019).

[14] Wikipedia contributors, Bellman–Ford algorithm, Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Bellman%E2%80%93Ford_algorithm&oldid=873252848 (visitado el 15 de febrero de 2019).